Turing machines deciders, part I

The bbchallenge Collaboration*

Abstract

The Busy Beaver Challenge (or bbchallenge) aims at collaboratively solving the following conjecture: "S(5) = 47,176,870" [Radó, 1962], [Marxen and Buntrock, 1990], [Aaronson, 2020]. This conjecture says that if a 5-state Turing machine runs for more than 47,176,870 steps without halting then it will never halt (starting from all-0 memory tape). Proving this conjecture amounts to decide whether or not 181,385,789 Turing machines with 5 states halt or not – starting from all-0 tape [11]. In order to decide the behavior of these machines we write *deciders*, programs that takes as input a Turing machine and output either `HALT`, `NONHALT`, or `UNKNOWN`. Each decider is specialised in recognising a particular type of behavior that can be decided.

After two years of work, the Busy Beaver Challenge achieved its goal in July 2024 by delivering a proof of "S(5) = 47,176,870" formalised in Coq[1] [11]. In this document, we present deciders that were developed before the Coq proof and which did not end up being used in the proof[2]; nonetheless, they are relevant techniques for analysing Turing machines. Part II of this work is the decider section of our paper showing "S(5) = 47,176,870" [11], presenting the deciders that were used in the Coq proof.

Contents

*`bbchallenge.org`, email: `bbchallenge@bbchallenge.org`

[1] Renamed Rocq.

[2] Apart from the verifier part of Finite Automata Reduction, Section 6, which is used in [11].

1 Conventions

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

Table 1: Transition table of the current 5-state busy beaver champion: it halts after 47,176,870 steps. `https://bbchallenge.org/1RB1LC_1RC1RB_1RD0LE_1LA1LD_---0LA&status=halt`

The set $\mathbb{N}$ denotes $\{0, 1, 2 \dots\}$.

Turing machines. The Turing machines that are studied in the context of bbchallenge use a binary alphabet and a single bi-infinite tape. Machine transitions are either undefined (in which case the machine halts) or given by (a) a symbol to write (b) a direction to move (right or left) and (c) a state to go to. Table 1 gives the transition table of the current 5-state busy beaver champion. The machine halts after 47,176,870 steps (starting from all-0 tape) when it reads a 0 in state E, which is undefined.

A *configuration* of a Turing machine is defined by the 3-tuple: (i) state (ii) position of the head (iii) content of the memory tape. In the context of bbchallenge, *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 $c_1 \vdash_{\mathcal{M}} c_2$ 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 $c_1 \vdash^s c_2$ denote a sequence of s computation steps, and let $c_1 \vdash^* c_2$ denote zero or more computation steps. We write $c_1 \vdash \perp$ if the machine halts after executing one computation step from configuration c_1. In the context of bbchallenge, 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.

When discussing concrete configurations, we write $0^\infty\ s_1\ \cdots\ s_{k-1}\ [s_k]_q\ s_{k+1}\ \cdots\ s_n\ 0^\infty$ to mean the configuration where the machine is in state q, with the head positioned on the symbol s_k, and the tape both starts and end by an infinite sequence of 0s, represented 0^∞. Thus, the initial configuration of the machine can be written as $0^\infty\ [0]_A\ 0^\infty$.

Directional Turing machines. We will sometimes prefer to think of the tape head as being between symbols. Thus, we write $l \overset{q}{\triangleleft} r$, with $l, r \in \{0,1\}^*$ to mean that the head is at the rightmost symbol of l, and $l \overset{q}{\triangleright} r$ to mean that the head is at the leftmost symbol of r. For example, $0^\infty\ [1]_A\ 0^\infty$ can be written as $0^\infty\ 1 \overset{A}{\triangleleft} 0^\infty$ or $0^\infty \overset{A}{\triangleright} 1\ 0^\infty$.

Space-time diagram. 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 (black is 0 and white is 1).
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.

2 Cyclers

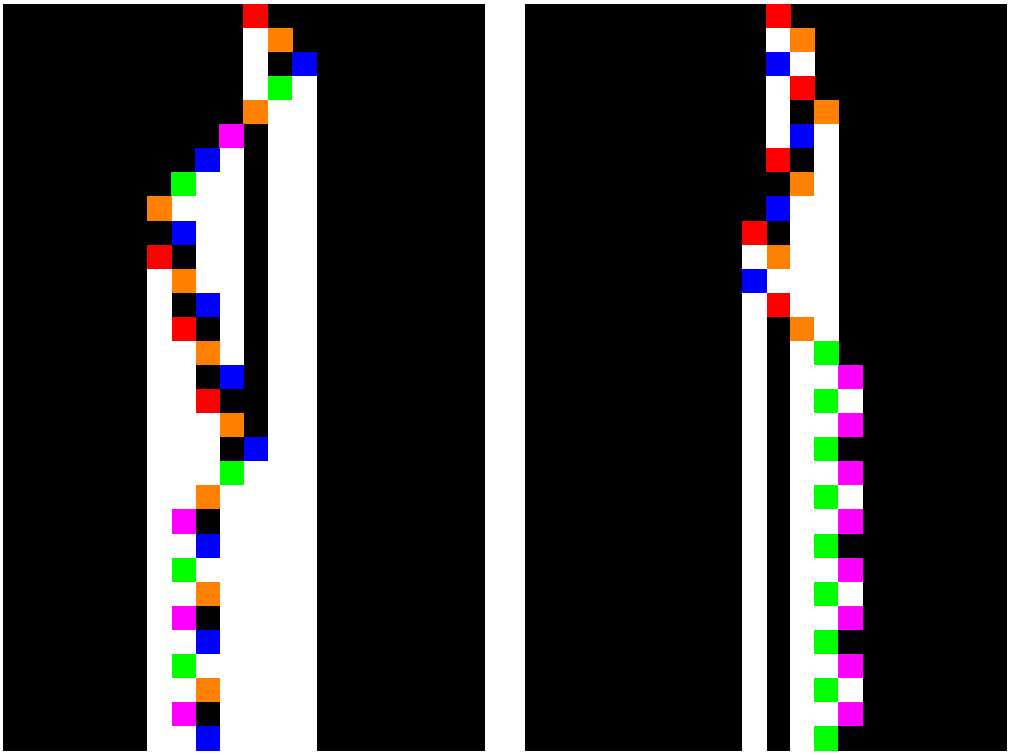


Figure 1: Space-time diagrams of the 30 first steps of bbchallenge's machines #279,081 (left) and #4,239,083 (right) which are both "Cyclers": they eventually repeat the same configuration for ever. Access the machines at `https://bbchallenge.org/279081` and `https://bbchallenge.org/4239083`.

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[3].

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 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.

2.2 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

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

Algorithm 1 DECIDER-CYCLERS

```
1:  struct Configuration {
2:      State state
3:      int headPosition
4:      int → int tape
5:  }
6:
7:  procedure bool DECIDER-CYCLERS(TM machine,int timeLimit)
8:      Configuration currConfiguration = {.state = A, .headPosition = 0, .tape = {0:0}}
9:      Set<Configuration> configurationsSeen = {}
10:     int currTime = 0
11:     while currTime ≤ timeLimit do
12:         if currConfiguration in configurationsSeen then
13:             return true
14:         configurationsSeen.insert(currConfiguration)
15:         currConfiguration = TuringMachineStep(machine,currConfiguration)
16:         currTime += 1
17:         if currConfiguration == nil then
18:             return false //machine has halted, it is not a Cycler
19:     return false
```

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). □

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. □

2.3 Implementation

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

3 Translated cyclers

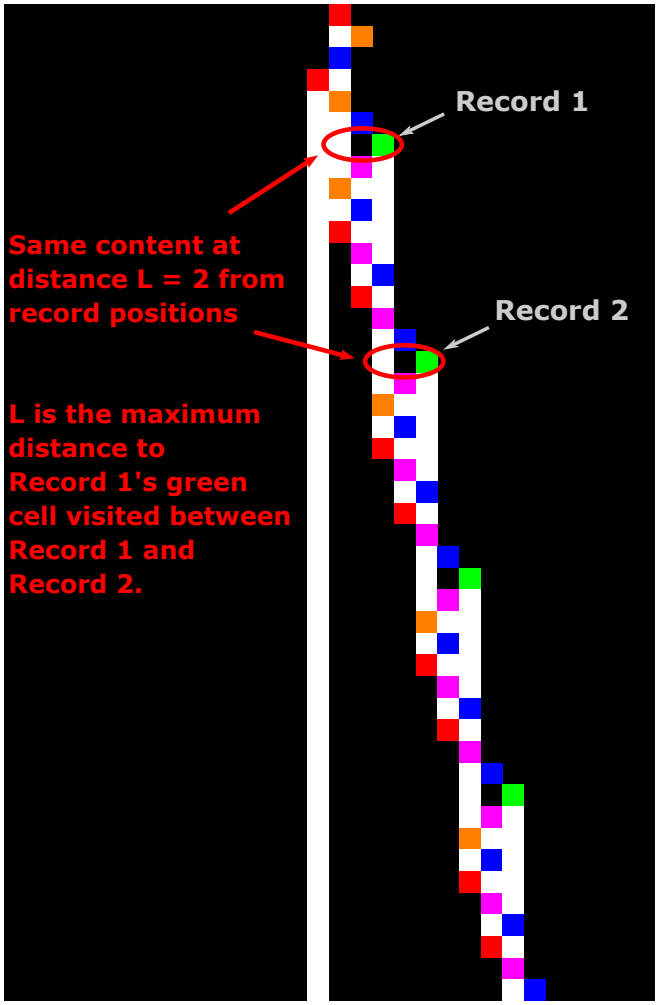


Figure 2: Example "Translated cycler": 45-step space-time diagram of bbchallenge's machine #44,394,115. See `https://bbchallenge.org/44394115`. The same bounded pattern is being translated to the right forever. The text annotations illustrate the main idea for recognising "Translated Cyclers": find two 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) 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. Similar ideas have been developed in [7].

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.

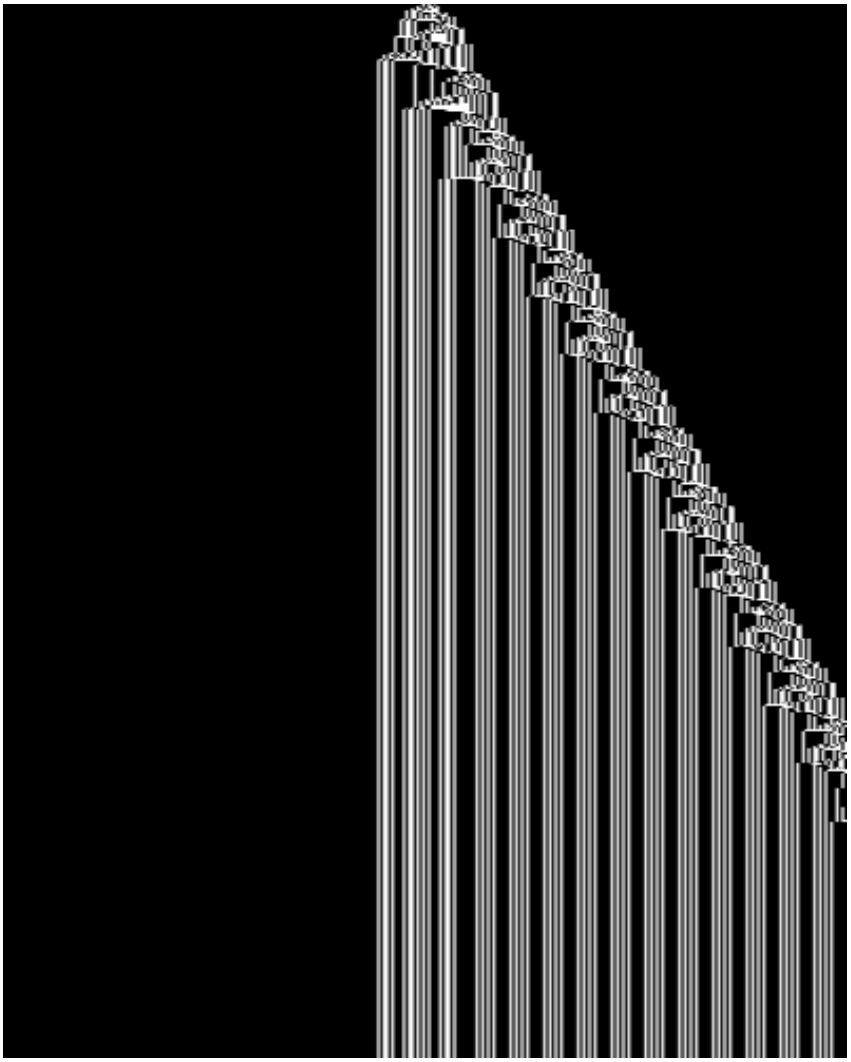


Figure 3: More complex "Translated cycler": 10,000-step space-time diagram (no state colours) of bbchallenge's machine #59,090,563. See `https://bbchallenge.org/59090563`.

3.1 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 3.2) 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

```
1: const int RIGHT, LEFT = 0, 1
2: struct ValueAndLastTimeVisited {
3:     int value
4:     int lastTimeVisited
5: }
6: struct Configuration {
7:     State state
8:     int headPosition
9:     int → ValueAndLastTimeVisited tape
10: }
11:
12: procedure bool DECIDER-TRANSLATED-CYCLERS(TM machine,int timeLimit)
13:     Configuration currConfiguration = {.state = A, .headPosition = 0, .tape = {0:{.value = 0, .lastTimeVisited = 0}}}
14:     // 0: right records, 1: left records
15:     List<Configuration> recordBreakingConfigurations[2] = [[],[]]
16:     int extremePositions[2] = [0,0]
17:     int currTime = 0
18:     while currTime < timeLimit do
19:         int headPosition = currConfiguration.headPosition
20:         currConfiguration.tape[headPosition].lastTimeVisited = currTime
21:         if headPosition > extremePositions[RIGHT] or headPosition < extremePositions[LEFT] then
22:             int recordSide = (headPosition > extremePositions[RIGHT]) ? RIGHT : LEFT
23:             extremePositions[recordSide] = headPosition
24:             if AUX-CHECK-RECORDS(currConfiguration, recordBreakingConfigurations[recordSide], recordSide) then
25:                 return true
26:             recordBreakingConfigurations[recordSide].append(currConfiguration)
27:         currConfiguration = TuringMachineStep(machine,currConfiguration)
28:         currTime += 1
29:         if currConfiguration == nil then
30:             return false //machine has halted, it is not a Translated Cycler
31:     return false
```

3.2 Correctness

Definition 3.1 (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 3.2 (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

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

```
1: procedure int COMPUTE-DISTANCE-L(Configuration currRecord, Configuration olderRecord,
   int recordSide)
2:     int olderRecordPos = olderRecord.headPosition
3:     int olderRecordTime = olderRecord.tape[olderRecordPos].lastTimeVisited
4:     int currRecordTime = currRecord.tape[currRecord.headPosition].lastTimeVisited
5:     int distanceL = 0
6:     for int pos in currRecord.tape do
7:         if pos > olderRecordPos and recordSide == RIGHT then continue
8:         if pos < olderRecordPos and recordSide == LEFT then continue
9:         int lastTimeVisited = currRecord.tape[pos].lastTimeVisited
10:        if lastTimeVisited ≥ olderRecordTime and lastTimeVisited ≤ currRecordTime then
11:            distanceL = max(distanceL,abs(pos-olderRecordPos))
12:    return distanceL
13:
14: procedure bool AUX-CHECK-RECORDS(Configuration currRecord, List<Configuration> older-
   Records, int recordSide)
15:    for Configuration olderRecord in olderRecords do
16:        if currRecord.state != olderRecord.state then
17:            continue
18:        int distanceL = COMPUTE-DISTANCE-L(currRecord,olderRecord,recordSide)
19:        int currExtremePos = GET-EXTREME-POSITION(currRecord.tape,recordSide)
20:        int olderExtremePos = GET-EXTREME-POSITION(olderRecord.tape,recordSide)
21:        int step = (recordSide == RIGHT) ? -1 : 1
22:        bool isSameLocalTape = true
23:        for int offset = 0; abs(offset) ≤ distanceL; offset += step do
24:            if currRecord.tape[currExtremePos+offset].value !=
   olderRecord.tape[olderExtremePos+offset].value then
25:                isSameLocalTape = false
26:                break
27:        if isSameLocalTape then
28:            return true
29:    return false
```

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 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 3.3. 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 3.2). 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 3.2, 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 3.2). 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 3.4. Let $\mathcal{M}$ be a Turing machine and t a time limit. The conditions of Lemma 3.3 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 3.3. 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 3.2) 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).

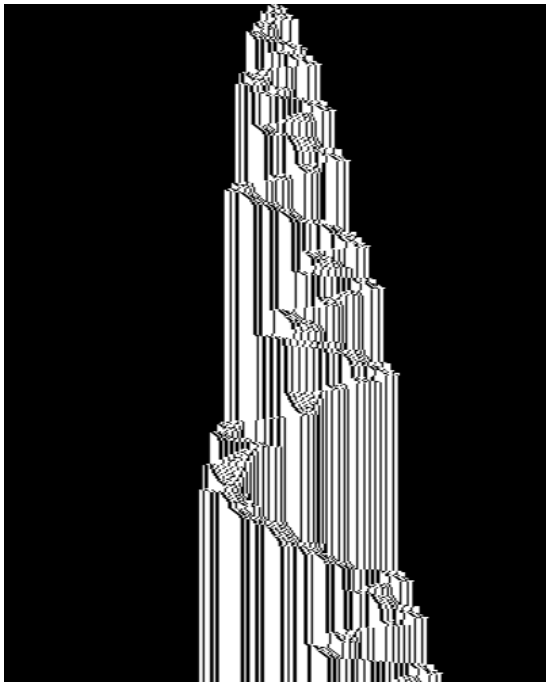
Thanks to the computation of COMPUTE-DISTANCE-L the routine AUX-CHECK-RECORDS is able to check whether the tape content at distance L of the record-breaking position in both record-holding configurations is the same or not (Algorithm 3, l.23). The routine returns `true` if they are the same and the function 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 3.3, which is what we wanted. □

Corollary 3.5. 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 3.3 and Theorem 3.4. □

3.3 Implementation

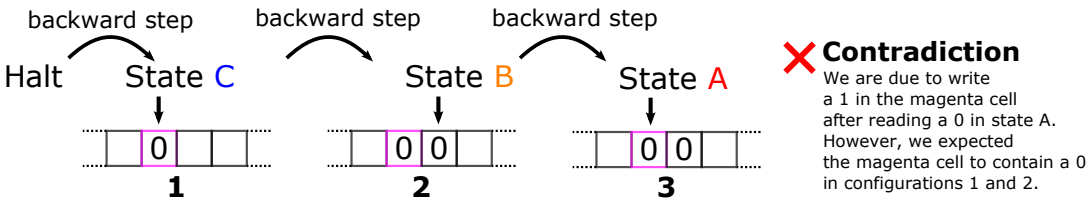
The decider was coded in `golang` and is accessible at this link: `https://github.com/bbchallenge/bbchallenge-deciders/tree/main/decider-translated-cyclers`.



(a) 10,000-step space-time diagram of bbchallenge's machine #55,897,188. `https://bbchallenge.org/55897188`

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

(b) Transition table of machine #55,897,188.



(c) Contradiction reached after 3 backward steps: machine #55,897,188 cannot reach its halting configuration hence it does not halt.

Figure 4: Applying backward reasoning on bbchallenge's machine #55,897,188. (a) 10,000-step space-time diagram of machine #55,897,188. The *forward* behavior of the machine looks very complex. (b) Transition table. (c) We are able to deduce that the machine will never halt thanks to only 3 backward reasoning steps: because a contradiction is met, it is impossible to reach the halting configuration in more than 3 steps – and, by (a), the machine did not halt in 10,000 steps starting from all-0 tape.

4 Backward Reasoning

Backward reasoning, as described in [8], takes a different approach than what has been done with deciders in Sections 2 and 3. Indeed, instead of trying to recognise a particular kind of machine's behavior, the idea of backward reasoning is to show that, independently of the machine's behavior, the halting configurations are not reachable. In order to do so, the decider simulates the machine *backwards* from halting configurations until it reaches some obvious contradiction.

Figure 4 illustrates this idea on bbchallenge's machine #55,897,188. From the space-time diagram, the *forward* behavior of the machine from all-0 tape looks to be extremely complex, Figure 4a. However, by reconstructing the sequence of transitions that would lead to the halting configuration (reading a 0 in state C), we reach a contradiction in only 3 steps, Figure 4c. Indeed, the only way to reach state C is to come from the right in state B where we read a 0. The only way to reach state B is to come from the left in state A where we read a 0. However, the transition table (Figure 4b) is instructing us to write a 1 in that case, which is not consistent with the 0 that we assumed was at this position in order for the machine to halt.

Backward reasoning in the case of Figure 4 was particularly simple because there was only one possible

previous configuration for each backward step – i.e. there is only one transition that can reach state C and same for state B. In general, this is not the case and the structure created by backward reasoning is a tree of configurations instead of just a chain. If all the leaves of a backward reasoning tree of depth D reach a contradiction, we know that if the machine runs for D steps from all-0 tape then the machine cannot reach a halting configuration and thus does not halt.

4.1 Pseudocode

Algorithm 4 DECIDER-BACKWARD-REASONING

```
1: const int RIGHT, LEFT = 0, 1
2: struct Transition {
3:     State state
4:     int read, write, move
5: }
6: struct Configuration {
7:     State state
8:     int headPosition
9:     int → int tape
10:    int depth
11: }
12:
13: procedure Configuration APPLY-TRANSITION-BACKWARDS(Configuration conf,Transition t)
14:    int reversedHeadMoveOffset = (t.move == RIGHT) ? -1 : 1
15:    int previousPosition = conf.headPosition+reversedHeadMoveOffset
16:    if previousPosition in conf.tape and conf.tape[previousPosition] != t.write then
17:        return nil // Backward contradiction spotted
18:    Configuration previousConf = {.state = t.state, .headPosition = previousPosition, .tape =
   conf.tape, .depth = conf.depth + 1}
19:    previousConf.tape[previousPosition] = t.read
20:    return previousConf
21:
22: procedure bool DECIDER-BACKWARD-REASONING(TM machine,int maxDepth)
23:    Stack<Configuration> configurationStack
24:    for Pair<State, int> (state,read) in GET-UNDEFINED-TRANSITIONS(machine) do
25:        Configuration haltingConfiguration = {.state = state, .headPosition = 0, .tape = {0: read},
   .depth = 0}
26:        configurationStack.push(haltingConfiguration)
27:    while !configurationStack.empty() do
28:        Configuration currConf = configurationStack.pop()
29:        if currConf.depth > maxDepth then return false
30:        for Transition transition in GET-TRANSITIONS-REACHING-STATE(machine,currConf.state) do
31:            Configuration previousConf = APPLY-TRANSITION-BACKWARDS(currConf, transition)
32:            // If no contradiction
33:            if previousConf != nil then
34:                configurationStack.push(previousConf)
35:    return true
```

We assume that we are given routine GET-UNDEFINED-TRANSITIONS(machine) which returns the list of (state,readSymbol) pairs of all the undefined transitions in the machine's transition table, for instance [(C,0)] for the machine of Figure 4b. We also assume that we are given routine GET-TRANSITIONS-REACHING-STATE(machine,targetState) which returns the list of all machine's transitions that go to the specified target state, for instance [(A,1,0LD),(C,1,1LD),(D,1,1LD)] for target state D in the machine of Figure 4b. These two routines contain very minimal logic as they only lookup in the description of the machine for the required information.

4.2 Correctness

Theorem 4.1. Let $\mathcal{M}$ be a Turing machine and $D \in \mathbb{N}$. Then, DECIDER-BACKWARD-REASONING($\mathcal{M}$,D) returns `true` if and only if no undefined transition of $\mathcal{M}$ can be reached in more than D steps.

Proof. The tree of backward configurations is maintained in a DFS fashion through a stack (Algorithm 4, l.23). Initially, the stack is filled with the configurations where only one tape cell is defined and state is set such that the corresponding transition is undefined (i.e. the machine halts after that step), l.24-26.

Then, the main loop runs until either (a) the stack is empty or (b) one leaf exceeded the maximum allowed depth, l.27 and l.29. Note that running the algorithm with increased maximum depth increases its chances to contradict all branches of the backward simulation tree. At each step of the loop, we remove the current configuration from the stack and we try to apply all the transitions that lead to this configuration backwards by calling routine APPLY-TRANSITION-BACKWARDS(configuration, transition).

The only case where it is not possible to apply a transition backwards, i.e. the case where a contradiction is reached, is when the tape symbol at the position where the transition comes from (i.e. to the right if transition movement is left and vice-versa) is defined but is not equal to the write instruction of the transition. Indeed, that means that the future (i.e. previous backward steps) is not consistent with the current transition's write instruction. This logic is in l.16. Otherwise, we can construct the previous configuration (i.e. next backward step) and augment depth by 1. We then stack this configuration in the main routine (l.34).

The algorithm returns `true` if and only if the stack ever becomes empty which means that all leaves of the backward simulation tree of depth D have reached a contradiction and thus, no undefined transition of the machine is reachable in more than D steps. □

Corollary 4.2. Let $\mathcal{M}$ be a Turing machine and $D \in \mathbb{N}$. If DECIDER-BACKWARD-REASONING($\mathcal{M}$,D) returns `true` and machine $\mathcal{M}$ can run D steps from all-0 tape without halting then the behavior of $\mathcal{M}$ from all-0 tape has been decided: $\mathcal{M}$ does not halt.

Proof. By Theorem 4.1 we know that no undefined transition of $\mathcal{M}$ can be reached in more than D steps. Hence, if machine $\mathcal{M}$ can run D steps from all-0 tape without halting, it will be able to run the next $D+1^{\text{th}}$ step. From there, the machine cannot halt or it would contradict the fact that halting trajectories have at most D steps. Hence, $\mathcal{M}$ does not halt from all-0 tape. □

4.3 Implementation

The decider was coded in `golang` and is accessible at this link: `https://github.com/bbchallenge/bbchallenge-deciders/blob/main/decider-backward-reasoning`. Note that collaborative work allowed to find a bug in the initial algorithm that was implemented[4].

[4]Thanks to collaborators `https://github.com/atticuscull` and `https://github.com/modderme123`.

5 Halting Segment

Acknowledgement. Sincere thanks to bbchallenge's contributor Iijil who initially presented this method and the first implementation[5]. Other contributors have contributed to this method by producing alternative implementations (see Section 5.3) or discussing and writing the formal proof presented here: Mateusz Naściszewski (Mateon1), Nathan Fenner, Tony Guilfoyle, Justin Blanchard, Tristan Stérin (cosmo), and, Pavel Kropitz (uni).

5.1 Overview

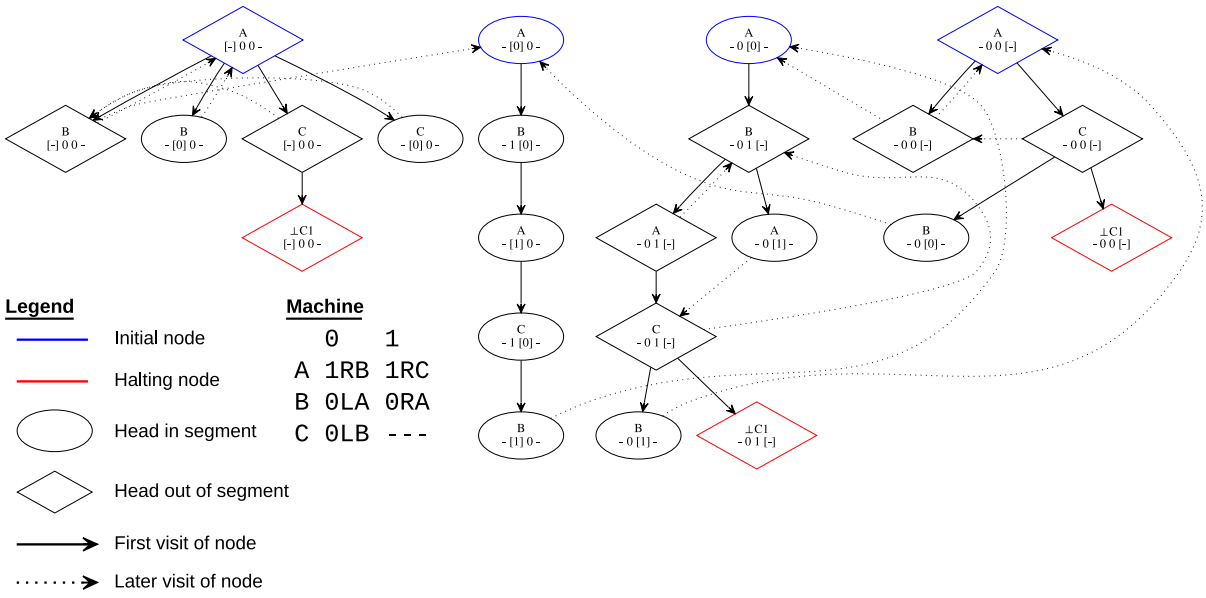


Figure 5: Halting Segment graph for the 3-state machine `https://bbchallenge.org/1RB1RC_0LAORA_0LB---` and segment size 2, see Definition 5.3. Nodes of this graph correspond to *segment configurations* (Definition 5.1), i.e. configurations of the machine on a finite segment (here, of size 2). In a node, the machine's head position is represented between brackets and the symbol `-` represents the outside of the segment (either to the left or to the right). Nodes where the machine's head is within the segment (circle shape) only one have child corresponding to the next step of the machine and nodes where the head is outside of the segment (diamond shape) may have multiple children corresponding to all the theoretically possible ways (deduced from the machine's transition table) that the machine can enter the segment back or continue to stay out of it. In order to improve readability, edges that revisit a node are dotted. The machine presented here does not halt because the halting nodes (red outline) that are reachable from the initial nodes (blue outline) do not cover all the positions of the segment (there is no halting node for any of the two internal positions of the segment), by contraposition of Theorem 5.4.

The idea of the Halting Segment technique is to simulate a Turing machine on a finite segment of tape. When the machine leaves the segment in a certain state, we consider all the possible ways that it can re-enter the segment or stay out of it, based on the machine's transition table. For a given machine and segment size, this method naturally gives rise to a graph, the Halting Segment graph (formally defined in Definition 5.3).

Figure 5 gives the Halting Segment graph of the 3-state machine[6] `https://bbchallenge.org/1RB1RC_0LAORA_0LB---` for segment size 2. Let's describe this graph in more details:

- Nodes correspond to *segment configurations* (Definition 5.1), i.e. the state in which the machine is together with the content of the segment and the position of the head in the segment (or outside of it). For instance, the leftmost node in blue and diamond shape in Figure 5 is `A [-] 0 0 -` which means that the machine is in state A, that the segment currently contains `0 0` and that the machine's head is currently outside of the segment, to the left of it.

[5]See: `https://discuss.bbchallenge.org/t/decider-halting-segment`.
[6]We chose a 3-state machine in order to have a graph of reasonable size.

- Initial nodes (blue outline) correspond to all segment configurations that match the initial configuration of the machine (all-0 tape and state A). There are $n+2$ initial nodes with n the size of the segment. Halting nodes (red outline) give the segment configurations where the machine has halted together with the halting transition that was used. For instance, in Figure 5, the leftmost halting node ⊥ `C1 [-] 0 0 -` signifies that the machine has halted (⊥), using halting transition `C1` (reading a 1 in state C), to the left of the segment which contains `0 0`.
- Nodes with a circle shape correspond to segment configurations where the tape's head is **inside** the segment. Such nodes only have one child, which corresponds to the next machine configuration.
- Nodes with a diamond shape correspond to segment configurations where the head is **outside** the segment. These nodes may have several children corresponding to all the ways that the head, in the current state, can stay outside of the segment or enter it back. For instance, the leftmost node in blue and diamond shape in Figure 5, `A [-] 0 0 -`, has 4 children: `B [-] 0 0 -` and `B - [0] 0 -` and `C [-] 0 0 -` and `C - [0] 0 -`. This is because the transitions of the machine in state A are `1RB` and `1RC` and that the move `R` allows either to enter the segment back or to continue being out of it (if the head is far from the segment's left frontier). Note that the write symbol `1` of the transitions are ignored since we do not keep track of the tape outside of the segment.
- In order to increase the readability of Figure 5, only one entrant edge for each node has been drawn with a solid line, corresponding to the first visit of that node in the particular order that the graph was visited. Later visits were drawn with a dotted line.

What is special about the Halting Segment graph? We show in Theorem 5.4 that if a machine halts, then, for all segment sizes, its Halting Segment graph contains a set of halting nodes (red outline), for the same halting transition, that covers the entire segment and its outside, i.e. such that there is at least one such node per segment's position and outside of it (left and right). By contraposition, if there is no set of covering halting nodes for a halting transition, the machine does not halt. In Figure 5, we deduce that machine `https://bbchallenge.org/1RB1RC_0LA0RA_0LB---` does not halt since the halting nodes of halting transition `C1` are ⊥ `C1 [-] 0 0 -`, ⊥ `C1 - 0 1 [-]` and ⊥ `C1 - 0 0 [-]` which does not cover the entire segment (both internal segment positions are not covered).

Interestingly, Halting Segment is the method that was used by Newcomb Greenleaf to prove[7] that Marxen & Buntrock's chaotic machine[8] [8] does not halt.

5.2 Formal proof

Definition 5.1 (Segment configurations)**.** Let the *segment size* be $n \in \mathbb{N}$. A *segment configuration* is a 3-tuple: (i) state, (ii) $w \in \{0,1\}^n$ which is the segment's content and (iii) the position of the machine's head is an integer $p \in [\![-1, n]\!]$ where positions $[\![0, n[\![$ correspond to the interior of the segment, position -1 for outside to the left and n for outside to the right. *Halting segment configurations* are segment configurations where the state is ⊥ and with an additional information (iv) of which halting transition of the machine has been used to halt.

Example 5.2. In Figure 5 we have $n = 2$ and, the leftmost node in blue and diamond shape corresponds to segment configuration `A [-] 0 0 -` (i) state A, (ii) $w =$ `00` and (iii) $p = -1$. The rightmost node in red and diamond shape corresponds to halting segment configuration ⊥ `C1 - 0 0 [-]` (i) state ⊥, (ii) $w =$ `00`, (iii) $p = 2$ and (iv) halting transition `C1`.

Definition 5.3 (Halting Segment graph)**.** Let $\mathcal{M}$ be a Turing machine and $n \in \mathbb{N}$ a segment size. The Halting Segment graph for $\mathcal{M}$ and n is a directed graph where the nodes are segment configurations (Definition 5.1). The graph is generated from $n+2$ *initial nodes* (blue outline in Figure 5) that are all in state A with segment content 0^n (n consecutive 0s) but where the head is at each of the $n+2$ possible positions, one per each initial node, see the blue nodes in Figure 5 for an example. Then, edges that go out of a given node r are defined as follows:

- If r's head position is inside the segment (circle nodes in Figure 5), then r only has one child corresponding to the next simulation step for machine $\mathcal{M}$. For instance, in Figure 5, node `A - [0] 0 -` has a unique child `B - 1 [0] -`, following machine's transition `A0` which is `1RB`. That child can be a halting segment configuration if the transition to take is halting.

[7] `http://turbotm.de/~heiner/BB/TM4-proof.txt`
[8] `https://bbchallenge.org/76708232`

- If r's head position is outside the segment (diamond nodes in Figure 5), then, we consider each transition of r's state. There are three cases:
  1. If the transition is halting, we add a child to r which is the halting segment configuration node corresponding to this transition. For instance, in Figure 5, `C [-] 0 0 -` has halting child ⊥ `C1 [-] 0 0 -` corresponding to halting transition `C1`.
  2. If the transition's movement goes further away from the segment (e.g. we are to the left of the segment, $p = -1$, and the transition movement is `L`), we add one child for this transition that only differs from its parent in the new state that it moves into. For instance, in Figure 5, `A - 0 0 [-]` has child `B - 0 0 [-]` for transition `A0` which is `1RB`.
  3. If the transition's movement goes in the direction of the segment (e.g. we are to the left of the segment, $p = -1$, and the transition movement is `R`), we add two children for this transition. One corresponding to the case where that movement is made at the border of the segment and allows to re-enter the segment and the other one corresponding to the case where that movement is made farther away from the border and does not re-enter yet. For instance, in Figure 5, node `A [-] 0 0 -` has children `B [-] 0 0 -` and `B - [0] 0 -` for transition `A0` which is `1RB`.

  Halting nodes are nodes corresponding to halting segment configurations (red outline in Figure 5).

Theorem 5.4 (Halting Segment)**.** Let $\mathcal{M}$ be a Turing machine and $n \in \mathbb{N}$ a segment size. Let G be the Halting Segment graph for $\mathcal{M}$ and n (Definition 5.3). If $\mathcal{M}$ halts in halting transition T when started from state A and all-0 tape, then G must contain a halting node for transition T for each of the $n + 2$ possible values of the head's position $p \in [\![-1, n]\!]$.

Proof. Consider the trace of configurations of $\mathcal{M}$ (full configurations, not segment configurations, as defined in Section 1) from the initial configuration (state A and all-0 tape) to the halting configuration which happens using halting transition T. Starting from the halting configuration, construct the halting segment configuration (with segment size n) for T using any position $p \in [\![-1, n]\!]$ in the segment and fill the segment's content from what is written on the tape around the head in the halting configuration of $\mathcal{M}$. From there, work your way up to the initial configuration: at each step construct the associated segment configuration. This sequence of segment configurations constitute a set of nodes in the Halting Segment graph G of $\mathcal{M}$ for segment size n such that each node points to the next one. At the top of that chain there will be a node matching the initial configuration: state A, all-0 segment and head position somewhere in $[\![-1, n]\!]$, i.e. an initial node.

Hence we have shown that all halting nodes for transition T for each of the $n + 2$ possible values of the head's position $p \in [\![-1, n]\!]$ are reachable from some initial node(s). □

Remark 5.5. By contraposition of Theorem 5.4, if, for all halting transitions T there is at least one halting node (red outline in Figure 5) for some position in the segment that is not reachable from one of the initial node (blue outline in Figure 5) then the machine does not halt. That way, in Figure 5, we can conclude that machine `https://bbchallenge.org/1RB1RC_0LA0RA_0LB---` does not halt since the halting nodes of halting transition `C1` are ⊥ `C1 [-] 0 0 -`, ⊥ `C1 - 0 1 [-]` and ⊥ `C1 - 0 0 [-]` which does not cover the entire segment (both internal segment positions are not covered).

Note that if all of the segment's positions are covered for some halting transition, we cannot conclude that the machine does not halt, but it does not mean that the machine necessarily halts either.

Remark 5.6. Some non-halting machines cannot be decided using Halting Segment for any segment size. Such a machine is for instance `https://bbchallenge.org/1RB---_1LC0RB_1LB1LA`.

5.3 Implementations

Here are the implementations of the method that were realised. Almost all of them construct the Halting Segment graph from the halting nodes (backward implementation) instead of from the initial nodes (forward implementation):

1. Iijil's who originally proposed the method, `https://github.com/bbchallenge/bbchallenge-deciders/tree/main/decider-halting-segment`, and was independently reproduced by Tristan Stérin (cosmo) `https://github.com/bbchallenge/bbchallenge-deciders/tree/main/decider-halting-segment-re` (backward implementation)

2. Mateusz Naściszewski (Mateon1)'s: `https://gist.github.com/mateon1/7f5e10169abbb50d1537165c6e71733b` (forward implementation)

3. Nathan Fenner's which has the interesting feature of being written in a language for formal verification (Dafny): `https://github.com/Nathan-Fenner/bbchallenge-dafny-deciders/blob/main/halting-segment.dfy` (backward implementation)

4. Tony Guilfoyle: `https://github.com/TonyGuil/bbchallenge/tree/main/HaltingSegments` (backward implementation)

Iijil's implementation (1) is a bit different from what is presented in this document because the Halting Segment graph is constructed backward (i.e. from the halting nodes instead of from the initial nodes). Also, the method adopts a lazy strategy consisting in testing only odd segment sizes (up to size $n_{\max}$) and placing the head's position at the center of the tape. Finally, the information of state is not stored for nodes where the head is outside the segment. These implementation choices make the implementation a bit weaker than what was presented here.

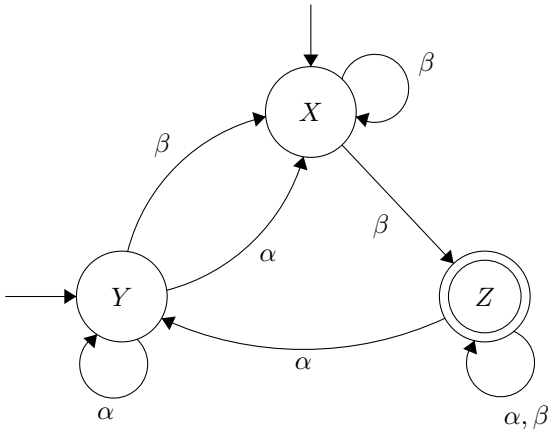


Figure 6: Example Nondeterministic Finite Automaton (NFA) with 3 states X, Y and Z, alphabet $\mathcal{A} = \{\alpha, \beta\}$, initial states X and Y, and accepting state Z. The linear-algebra representation of this NFA is given in Example 6.1. Example accepted words are: β, $\alpha\beta$, $\alpha\alpha\beta\beta$. Example rejected words are: α, $\alpha\alpha$, $\alpha\alpha\alpha$.

6 Finite Automata Reduction (FAR)

Acknowledgement. Sincere thanks to bbchallenge's contributor Justin Blanchard who initially presented this method and the first implementation[9]. Others have contributed to this method by producing alternative implementations (see Section 6.7) or discussing and writing the formal proof presented here: Tony Guilfoyle, Tristan Stérin (cosmo), Nathan Fenner, Mateusz Naściszewski (Mateon1), Konrad Deka, Iijil, Shawn Ligocki.

6.1 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 [6] 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[10]
2. C is *closed* under transitions: for any $c \in C$, the configuration one step later belongs to C[10]
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 [6] [5].

Here, we develop an original *co-CTL* technique[11], 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).

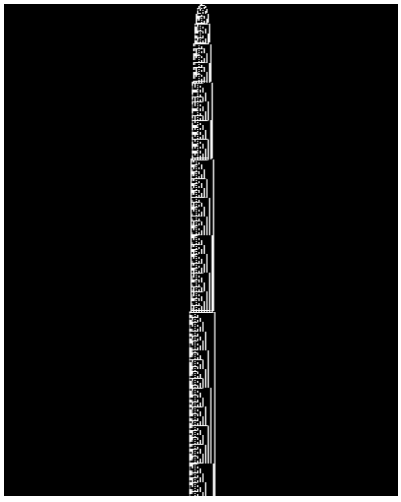
One important aspect of the technique is that, given a Turing machine and its constructed NFA—if found—it is a computationally simple task to verify that the NFA's language does indeed recognise all eventually-halting configurations (with finitely many 1s) of the machine.

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

[10] Criteria 1–2 give a strict definition; in [6], 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.

[11] 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.

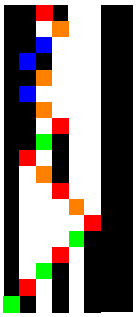
[12] `https://bbchallenge.org/1RB0LD_1LC1RA_0RB0LC_---1LA`, the machine exhibits a non-trivial counting behavior.



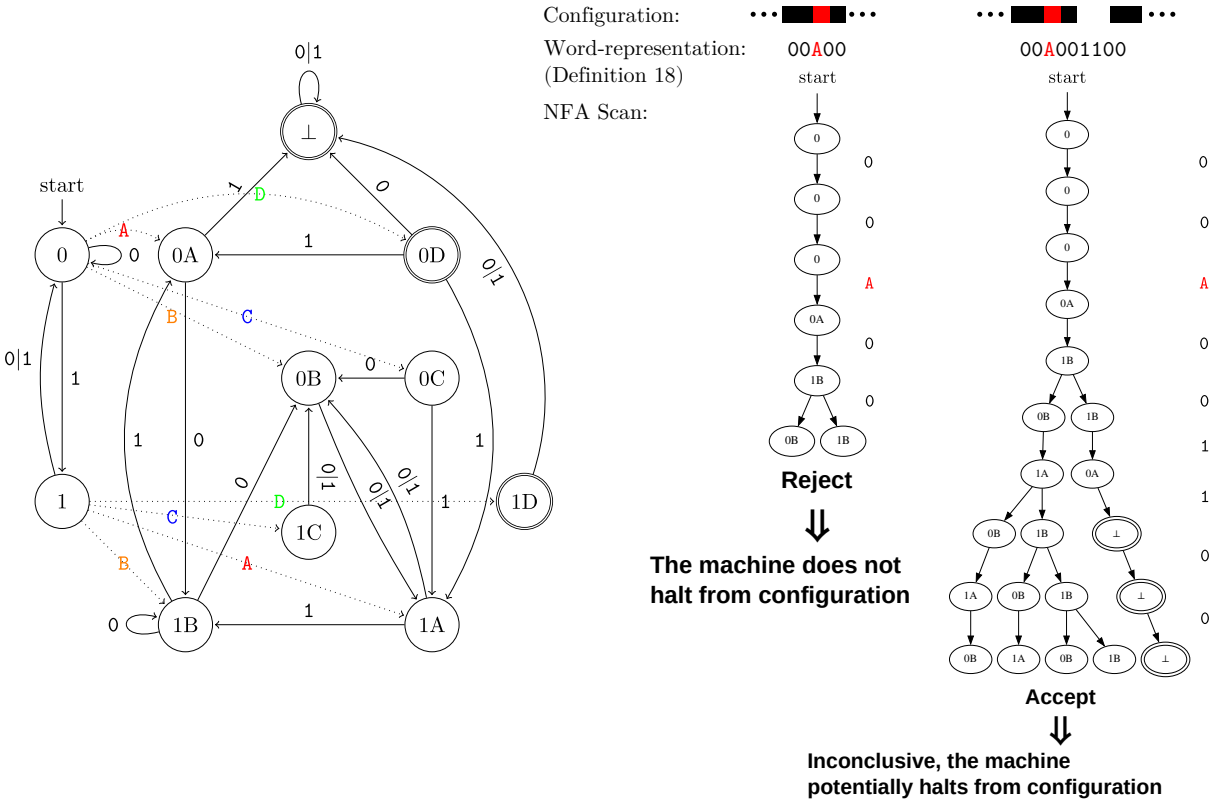
(a) 10,000-step space-time diagram of the 4-state Turing machine given in (b) from the all-0 initial configuration. The machine does not halt from the all-0 configuration.

	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 7: A Nondeterministic Finite Automaton, used as follows to decide a 4-state Turing machine[12]: (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 6.3), 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 6.2), 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.

6.2 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 7 gives a potential-halt-recognizing NFA for a 4-state Turing machine, constructed using the results of Section 6.3.

Let's first recall how Nondeterministic Finite Automata (**NFA**) can be described using linear algebra. Let $\mathbf{2}$ denote the Boolean semiring[13] $\{0,1\}$ with operations $+$ and $\cdot$ respectively implemented by OR and AND [4]. Let $\mathcal{M}_{m,n}$ 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 $q_0 \in \mathcal{M}_{1,n}$ 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 6.1. The NFA depicted in Figure 6, 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 1. Then, a word-representation of a configuration is defined by:

Definition 6.2 (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 6.3. A word-representation of the configuration on the top row of Figure 7(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 $T_0 \in \mathcal{M}_{n,n}$:

$$q_0 T_0 = q_0 \tag{6.1}$$
$$T_0 a^T = a^T \tag{6.2}$$

Note that Condition 6.2, $T_0 a^T = a^T$, 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.

[13] 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:

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

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 6.2. 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:

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

Which algebraically becomes:

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

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 $(f, r) \to \perp$ a halting transition.

Combining the above, we get our main result:

Theorem 6.4. 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:

$$q_0 T_0 = q_0 \qquad \text{(leading zeros ignored)} \tag{6.1}$$
$$T_0 a^T = a^T \qquad \text{(trailing zeros ignored)} \tag{6.2}$$
$$sa^T = 1 \qquad (s \text{ is accepted}) \tag{6.3}$$
$$sT_0, sT_1 \succeq s \qquad (s \text{ is a steady state}) \tag{6.4}$$
$$\forall u \in \{0,1\}^* : q_0 T_u T_f T_r \succeq s \qquad \text{if } (f, r) \to \perp \text{ is a halting transition of } \mathcal{M} \tag{6.5}$$
$$\forall b \in \{0,1\} : T_b T_f T_r \succeq T_t T_b T_w \qquad \text{if } (f, r) \to (t, w, \text{left}) \text{ is a transition of } \mathcal{M} \tag{6.6}$$
$$T_f T_r \succeq T_w T_t \qquad \text{if } (f, r) \to (t, w, \text{right}) \text{ is a transition of } \mathcal{M} \tag{6.7}$$
$$q_0 T_A a^T = 0 \qquad \text{(initial configuration rejected)} \tag{6.8}$$

Proof. Conditions (6.1)–(6.7) ensure that the NFA's language includes at least all eventually halting configurations of $\mathcal{M}$. Condition (6.8) ensures that the initial all-0 configuration of the machine is rejected, hence not eventually halting. Hence, if conditions (6.1)–(6.8) are satisfied, we can conclude that $\mathcal{M}$ does not halt from the initial all-0 configuration. □

Remark 6.5 (Verification)**.** Theorem 6.4 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 (6.1)–(6.8) hold and thus that the TM does not halt, is computationally simple[14]. Verifiers have been implemented for Theorem 6.4, see Section 6.7.

Now, we want to design an efficient search algorithm that will, for a given TM, try to find an NFA satisfying Theorem 6.4. 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 (6.1)–(6.7) is automatically satisfied by these NFAs.

[14]Note that although equation (6.5) 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 (6.5) replacing $q_0 T_u$ per such state.

6.3 Search algorithm: direct FAR algorithm

We design an efficient search algorithm for Theorem 6.4 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 6.6. The structure described above is followed by the NFA depicted in Figure 7(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 7 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 6.4.

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 (6.1) of Theorem 6.4.
3. Row vectors of matrices M_f (with $f \in \{A, \ldots, E\}$) are the standard basis row vectors e_0, ..., $e_{5l-1} \in \mathcal{M}_{1,d}$ 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 6.4 (6.1)–(6.7) for R_0, R_1, and a. If Condition (6.8) is also satisfied then, by Theorem 6.4, 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 6.4 conditions (6.4)–(6.7) 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[15].

$$R_r \succeq (e_\perp)^T e_\perp \qquad \text{for } r \in \{0,1\} \tag{6.4'}$$

$$\forall i \in [l] : R_r \succeq \text{row}_i(M_f)^T e_\perp \qquad \text{if } (f,r) \to \perp \text{ is a halting transition of } \mathcal{M} \tag{6.5'}$$

$$\forall b \in \{0,1\}, \forall i \in [l] : R_r \succeq \text{row}_{\delta(i,b)}(M_f)^T \text{row}_i(M_t) R_b R_w \qquad \text{if } (f,r) \to (t,w,\text{left}) \text{ is a transition of } \mathcal{M} \tag{6.6'}$$

$$\forall i \in [l] : R_r \succeq \text{row}_i(M_f)^T \text{row}_{\delta(i,w)}(M_t) \qquad \text{if } (f,r) \to (t,w,\text{right}) \text{ is a transition of } \mathcal{M} \tag{6.7'}$$

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

Lemma 6.7. There's a unique minimal solution (w.r.t $\preceq$) to the system of inequalities (6.4')–(6.7') and an effective way to compute it: initialize R_0, R_1 to zero, then set entries to 1 as (6.4'), (6.5') and (6.7') demand then iterate (6.6') until R_0 and R_1 stop changing.

Proof. First notice that (6.4'), (6.5') and (6.7') 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 (6.6') 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 (6.2), (6.3) of Theorem 6.4 translate to:

$$R_0 a^T = a^T \tag{6.2'}$$

$$a \succeq e_\perp \tag{6.3'}$$

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 (6.4'), 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 (6.2') and (6.3').

The last condition from Theorem 6.4 that we need to satisfy is (6.8) (rejection of the initial configuration), which translates to:

$$\mathrm{row}_0(M_A) a^T = 0 \tag{6.8'}$$

By minimality, a solution of (6.2') and (6.3') will satisfy (6.8') if and only if the minimal solution exhibited above does. Hence, we check (6.8') for the minimal a that we constructed and there are two cases:

- If a satisfies (6.8') then we have found an NFA satisfying Theorem 6.4 and we can conclude that the Turing machine does not halt from the all-0 initial configuration.
- If a does not satisfy (6.8') 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 6.4 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 6.7) 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 (6.8') 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 5.

6.4 Efficient enumeration of Deterministic Finite Automata

The direct FAR algorithm (Section 6.3 and Algorithm 5) relies on a procedure to enumerate Deterministic Finite Automata (DFA). We first recall the formal definition of DFAs then give an efficient algorithm (Algorithm 6) to enumerate them and to prune the search space early based on using Lemma 6.7 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. [10]

To put this definition in the linear-algebraic framework: identify $q_0 \in Q$ with $0 \in [n] := \{0, \dots, n-1\}$; 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 $(\delta(0,0), \delta(0,1), \dots, \delta(n-1,0), \delta(n-1,1))$.

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

Algorithm 5 DECIDER-FINITE-AUTOMATA-REDUCTION-DIRECT

```
1: procedure bool DECIDER-FINITE-AUTOMATA-DIRECT(TM machine, int n, bool left_to_right)
2:     if not left_to_right then switch all left-going transitions of the TM to right-going and vice versa
3:     Matrix<bool, 5 * n + 1, 5 * n + 1> R[2 * n + 1][2] = [[0, 0], . . . , [0, 0]]
4:     ColVector<bool, 5*n+1> aT[2*n+1] = 0 // aT for transpose as a is row vector in Section 6.3
5:     ▷ Basis vector indexing: for row_i(M_f) use index 5 * i + f, and for e_⊥, use index 5 * n.
6:     Initialize R[0] using (6.4') and (6.5')
7:     Initialize aT[0] = e_⊥^T
8:     procedure CheckResult CHECK(List<int> L)
9:         k := L.length
10:        R[k], aT[k] = R[k-1], aT[k-1]
11:        Increase R[k] using (6.7'), with (i, w) = divmod(k-1, 2)
12:        repeat
13:            Increase R[k] using (6.6'), restricted to 2 * i + b < k
14:        until R[k] stops changing
15:        repeat
16:            aT[k] = R[k][0] · aT[k]
17:        until aT[k] stops changing
18:        if row_0(M_A) · aT[k] ≠ 0 then return SKIP
19:        else if k == 2*n then return STOP
20:        else return MORE
21:    return SEARCH-DFA(check)
```

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 6.8. 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 (6.1) 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 $m_{2q-1} \ge q$.

□

Corollary 6.9. $\{t_k\}_{k=0}^{\ell}$ ($\ell < 2n$) 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 $m_{2q-1} \ge q$ (for all $2q - 1 \le \ell$).

Proof. If $\ell = 2n-1$, $\{m_k\}$ grows to exactly $n-1$ (since $m_{2(n-1)-1} \geq n-1$), and lemma 6.8 applies. Otherwise, we may extend the sequence with $t_{\ell+1} = \min(m_\ell + 1, n-1)$, the same conditions apply. □

So, Algorithm 6 searches such DFAs incrementally (avoiding partial DFAs already deemed unworkable).

Algorithm 6 SEARCH-DFA

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

2: procedure bool SEARCH-DFA(int n, function<List<int>, CheckResult> check)
Require: check(t) ≠ MORE if t is a complete (length-2n) table
3:     int k = 1, t[2 * n] = [0, ..., 0], m[2 * n] = [0, ..., 0]
4:     loop
5:         state = check(length-k prefix of t)
6:         if state == MORE then
7:             int q_new = m[k-1] + 1
8:             t[k] = (q_new < n and 2*q_new-1 == k) ? q_new : 0
9:         else if state == SKIP then
10:            repeat
11:                if k ≤ 1 then return false
12:                k -= 1
13:            until t[k] ≤ m[k-1] and t[k] < n-1
14:            t[k] += 1
15:        else return true
16:        m[k] = max(m[k-1], t[k])
17:        k += 1
```

6.5 Generality of the method

In the preceding sections, we started from the idea of a closed language of word-representations of TM configurations, made a series of simplifying assumptions, and obtained a search algorithm. This raises a question: if *any* regular language proves a given TM infinite by co-CTL argument[16], must a proof of the form used in Theorem 6.4, let alone in §6.3, exist?

The answer is yes, and we sketch the proof below. The following definitions and results aren't needed to prove this decider method's soundness—or outside of this subsection at all—but they justify using Remark 6.5 to build a universal (regular) CTL verification scheme. Historically, they were discovered together with Algorithm 5, and motivated its development.

Any closed language $\mathcal{L}$ classifies the binary words $w \in \{0,1\}^*$ by Nerode congruence: $w \sim_{\mathcal{L}} w'$ if for every $z \in \{0,1,A,\ldots,E\}^*$, $wz \in \mathcal{L} \iff w'z \in \mathcal{L}$. We may form a modified version of the Turing machine $\mathcal{M}$, herein called $\mathcal{M}/\sim_{\mathcal{L}}$, with the following semantics:

A *configuration* of $\mathcal{M}/\sim_{\mathcal{L}}$ is defined by the 3-tuple: (i) a state of $\mathcal{M}$, (ii) an equivalence class $[w]_{\sim_{\mathcal{L}}}$ of some $w \in \{0,1\}^*$ representing the (strictly) left-of-head portion of the tape, (iii) a finite word $w \in \{0,1\}^*$, representing the remainder of the tape. We additionally define one distinct configuration, named $\perp$, which represents the machine in a halted state.

Note that any finitely supported configuration c of $\mathcal{M}$ maps to a configuration $[c]$ of $\mathcal{M}/\sim_{\mathcal{L}}$, by sending the left-of-head contents to their equivalence class modulo $\sim_{\mathcal{L}}$. This is a many-to-one mapping.

The *transitions* of $\mathcal{M}/\sim_{\mathcal{L}}$ are the images of those of $\mathcal{M}$: that is, if $c_1 \vdash_{\mathcal{M}} c_2$, $[c_1] \vdash_{\mathcal{M}/\sim_{\mathcal{L}}} [c_2]$. Since $c_1 \mapsto [c_1]$ is a many-to-one mapping, this definition makes $\mathcal{M}/\sim_{\mathcal{L}}$ a nondeterministic machine. In case $c_1 \vdash_{\mathcal{M}} \perp$ (i.e., the $\mathcal{M}$-transition from c_1 is undefined), we also define $[c_1] \vdash_{\mathcal{M}/\sim_{\mathcal{L}}} \perp$.

For any configurations c_1, c_2 of $\mathcal{M}$, if $[c_1] \vdash_{\mathcal{M}} / \sim_{\mathcal{L}} [c_2]$ and a word-representation of c_2 is in $\mathcal{L}$, observe that a word-representation of c_1 is also in $\mathcal{L}$. This follows because the $\mathcal{M}/\sim_{\mathcal{L}}$-transition must come from some transition $c_1' \vdash_{\mathcal{M}} c_2'$ of $\mathcal{M}$, where $[c_1] = [c_1']$ and $[c_2] = [c_2']$. Now, c_2' has a word-representation which differs from one of c_2 only by substituting a Nerode-congruent prefix, so c_2' also has a word-representation in $\mathcal{L}$. By closure of $\mathcal{L}$, this is true of c_1', and similarly by Nerode congruence this is true of c_1.

Similarly, if $[c_1] \vdash_{\mathcal{M}} / \sim_{\mathcal{L}} \perp$, c_1 has a word-representation in $\mathcal{L}$.

Say that $\mathcal{M}/\sim_{\mathcal{L}}$ halts from its initial configuration (which is the image of $\mathcal{M}$'s initial configuration) if there exists a sequence of $\mathcal{M}/\sim_{\mathcal{L}}$-transitions from it to $\perp$. The point of this is: if $\mathcal{L}$ is a closed language

[16] As regular languages' complements are regular, this is the same as a regular CTL (in the strict sense of §6.1) existing.

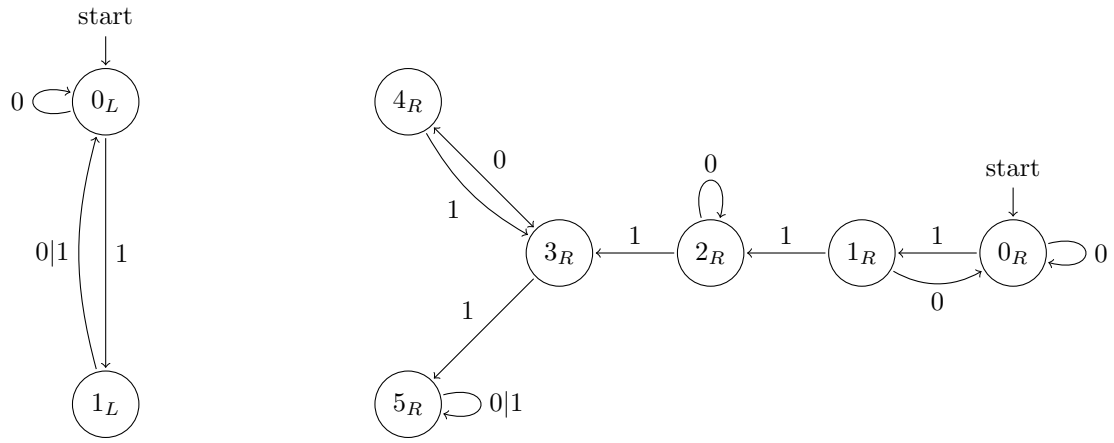


Figure 8: This pair of DFAs can also recognize halting configurations for the TM of figure 7. Configurations are classified by their head state, head bit, and the two half-tapes (processed outside-in by the DFAs.)

for $\mathcal{M}$, separating its initial configuration from all halting configurations, then that's impossible! For that would imply a sequence $[c_0] \vdash_{\mathcal{M}/\sim_\mathcal{L}} \cdots \vdash_{\mathcal{M}/\sim_\mathcal{L}} [c_n] \vdash_{\mathcal{M}/\sim_\mathcal{L}} \bot$, where c_0 is the initial configuration of $\mathcal{M}$ and $\{c_i\}_{i=1}^n$ is a sequence of other configurations. By the above, this would imply that c_0 has a word-representation in $\mathcal{L}$, contrary to assumption that $\mathcal{L}$ provides a CTL proof that $c_0 \not\vdash^*_\mathcal{M} \bot$.

We now seek to recover $\mathcal{L}$, or another regular language which leads to a CTL proof, by studying the halting problem of $\mathcal{M}/\sim_\mathcal{L}$. In fact, the work has been done already: observe that, just as any Turing machine $\mathcal{M}$ is equivalent to a PDA equipped with two stacks (corresponding to the strict left-hand side of the tape and the rest of it), the machine $\mathcal{M}/\sim_\mathcal{L}$ is equivalent to a standard nondeterministic PDA. (The control-state space of the PDA to is simply $\{0,1\}^*/\sim_\mathcal{L} \times \{A, \ldots, E\}$—which is a finite set by the Myhill-Nerode theorem. A transition of $\mathcal{M}/\sim_\mathcal{L}$ corresponding to a leftward TM transition pushes the written bit onto the stack. A transition of $\mathcal{M}/\sim_\mathcal{L}$ corresponding to a rightward TM transition pops the read bit off the stack.) The halting problem of a PDA is solved in [2]: the eventually-halting configurations of any PDA are in fact described by a regular language, whose construction corresponds exactly to the procedure of §6.3.

In summary: we may take the Myhill-Nerode DFA for the original language $\mathcal{L}$, restrict it to the alphabet $\{0,1\}$, apply the construction from §6.3 to obtain an NFA which recognizes precisely the halting configurations of $\mathcal{M}/\sim_\mathcal{L}$, and combine the DFA/NFA to form a recognizer for some closed language $\mathcal{L}'$ for $\mathcal{M}$; that is, it satisfies (6.1)–(6.7). We also know that a language solving the halting problem of $\mathcal{M}/\sim_\mathcal{L}$ rejects its initial configuration, and so (6.8) is also satisfied and the constructed $\mathcal{L}'$ provides a CTL proof for $\mathcal{M}$.

6.6 Search algorithm II: meet-in-the-middle DFA

A symmetric recognizer construction has also shown good results. Again, pass the left half-tape through a DFA with l states. Imagine a DFA with d states scanning the (strict) right half-tape right-to-left.

Remark 6.10. Our definitions require a left-to-right scan direction. Any NFA $(e_0, \{R_b\})$ can be transposed. (Transposing transition matrices reverses the arrows in the diagram, as with graph adjacency matrices.) We can shoehorn this into the preceding framework by making an accept state from R's transposed initial state e_0^T, defining middle transitions M_{fr} for the configuration's head state/bit, superposing all states of R to get our s vector, and trying to satisfy conditions like $e_0 M_{A0} e_0^T = 0$, $M_{fr} = \sum_L e_q^T s$ (for halt rules), $L_b M_{fr} \succeq M_{tb} R_w^T$ (for left rules), $M_{fr} R_b^T \succeq L_w M_{tb}$ (for right rules). What follows is more intuitive.

As in Figure 8, let's consider the DFAs on their own terms. Each one partitions its input into a family of regular languages (one per state). Accounting for the head state/bit and right half-tape, we obtain $l \cdot 5 \cdot 2 \cdot d$ classes of TM configuration. Propose a recognizer which distils this classification into a result. We'll work out conditions for a good "accepted" set $A \subseteq [l] \times \{A, \ldots, E\} \times \{0,1\} \times [d]$. If they're satisfiable, even if we don't prove the scheme sound, we can feed the left DFA into Algorithm 5 to check the result.

Despite the new setting, we can write out closure conditions analogous to §6.2's, each $\forall i \in [l], j \in [d]$:

$$(i, f, r, j) \in A \qquad \text{if } (f, r) \to \bot \text{ is a halting transition of } \mathcal{M} \tag{6.5"}$$

$$\forall b \in \{0,1\},\ (i, t, b, \delta_R(j, w)) \in A \implies (\delta_L(i, b), f, r, j) \in A \quad \text{if } (f, r) \to (t, w, \text{left}) \text{ is a transition of } \mathcal{M} \tag{6.6"}$$

$$\forall b \in \{0,1\},\ (\delta_L(i, w), t, b, j) \in A \implies (i, f, r, \delta_R(j, b)) \in A \quad \text{if } (f, r) \to (t, w, \text{right}) \text{ is a transition of } \mathcal{M} \tag{6.7"}$$

The goal, analogous to (6.8), is $(0, A, 0, 0) \notin A$.

We could now search all DFA pairs, checking if the smallest A closed under (6.5")–(6.7") rejects $(0, A, 0, 0)$. However, to get decent performance, we must express the above as a boolean satisfiability (SAT) problem.

Other lessons learned in practice: it was most effective to use the same state count on both sides ($l = d = n$), and it was decisively faster to impose the canonical form restrictions of Lemma 6.8.

Algorithm 7 shows how this works. Here especially, actual code can vary from the given pseudocode:

- If SAT solvers use integers for literals (variables and their negations), one needn't "allocate variables".
- It may be possible to simplify by adding propositional variables for more edge cases.
- The "outcomes are mutually exclusive" condition may be represented differently.
- Checking a solution is valid needn't involve Algorithm 5, if the author proves more.

6.7 Implementations

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 6.4 – i.e. programs that check that a given NFA gives a valid nonhalting proof for a given machine, see Remark 6.5 – 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`.

Algorithm 7 DECIDER-FINITE-AUTOMATA-REDUCTION-MITM-DFA

procedure bool DECIDER-FINITE-MITM-DFA(**TM** machine, **int** n)
  ▷ Allocate variables.
  Map<tuple, int> tk_eq, tk_le, mk_eq, A
  for all $(\mathrm{lr}, \mathrm{k}, \mathrm{y}) \in [2] \times [n] \times [2 * n] \times [n + 1]$ **do**
    if (k, y) == (0, 0) **then** tk_eq[lr, k, y] = true
    else if $0 \leq y \leq \min(k, n - 1)$ **then** tk_eq[lr, k, y] = NEW-VARIABLE
    else tk_eq[lr, k, y] = false
    if $y \leq 0$ **then** tk_le[lr, k, y] = tk_eq[lr, k, y]
    else if $0 \leq y \leq \min(k - 1, n - 2)$ **then** tk_le[lr, k, y] = NEW-VARIABLE
    else tk_le[lr, k, y] = true
    if (k, y) == (2*n-1, n-1) **then** mk_eq[lr, k, y] = true
    else if not $\lceil \frac{k}{2} \rceil \leq y < \min(n, k + 1)$ **then** mk_eq[lr, k, y] = false
    else if $\min(n, k + 1) - ((k + 1)/2) \leq 1$ **then** mk_eq[lr, k, y] = true
    else mk_eq[lr, k, y] = NEW-VARIABLE
  for all $(\mathrm{i}, \mathrm{f}, \mathrm{r}, \mathrm{j}) \in [n] \times [5] \times [2] \times [n]$ **do**
    if (k, y) == (2*n-1, n-1) **then** A[i, f, r, j] = false
    else A[i, f, r, j] = NEW-VARIABLE
  ▷ Transition validity: outcomes are mutually exclusive.
  for all $(\mathrm{lr}, k, \mathrm{y}) \in [2] \times [2 * n] \times [n]$ **do**
    NEW-CLAUSE$(\mathrm{tk_eq}(\mathrm{lr}, k, y) \implies \mathrm{tk_le}(\mathrm{lr}, k, y))$
    NEW-CLAUSE$(\mathrm{tk_le}(\mathrm{lr}, k, y) \implies \mathrm{tk_le}(\mathrm{lr}, k, y + 1))$
    NEW-CLAUSE$(\mathrm{tk_eq}(\mathrm{lr}, k, y + 1) \implies \neg \mathrm{tk_le}(\mathrm{lr}, k, y))$
  ▷ Transition validity: an outcome occurs.
  for all $(\mathrm{lr}, k) \in [2] \times \{1, \ldots, 2 * n - 1\}$ **do** NEW-CLAUSE$(\bigvee_{y=0}^{\min(k, n-1)} \mathrm{tk_eq}(\mathrm{lr}, k, y))$
  ▷ Closure conditions.
  for all $(i, j, (f, r)) \in [n]^2 \times$HALT-RULES(machine) **do**
    NEW-CLAUSE$(A[i, f, r, j])$
  for all $(i, j, \mathrm{ib}, \mathrm{jw}, (f, r, w, L, t)) \in [n]^4 \times$LEFT-RULES(machine) **do**
    NEW-CLAUSE$(\mathrm{tk_eq}[L, i, b, \mathrm{ib}] \wedge \mathrm{tk_eq}[R, j, w, \mathrm{jw}] \wedge A[i, t, b, \mathrm{jw}] \implies A[\mathrm{ib}, f, r, j])$
  for all $(i, j, \mathrm{iw}, \mathrm{jb}, (f, r, w, R, t)) \in [n]^4 \times$RIGHT-RULES(machine) **do**
    NEW-CLAUSE$(\mathrm{tk_eq}[R, j, b, \mathrm{jb}] \wedge \mathrm{tk_eq}[L, i, w, \mathrm{iw}] \wedge A[\mathrm{iw}, t, b, j] \implies A[i, f, r, \mathrm{jb}])$
  ▷ DFA is in canonical form (Lemma 6.8).
  for all $(\mathrm{lr}, k) \in [2] \times \{1, \ldots, 2 * n - 1\}$ **do**
    for $m = \lfloor k/2 \rfloor, \ldots, \min(n, k)$ **do**
      NEW-CLAUSE$(\mathrm{mk_eq}(\mathrm{lr}, k - 1, m) \implies \mathrm{tk_le}(\mathrm{lr}, k, m + 1))$
      NEW-CLAUSE$(\mathrm{mk_eq}(\mathrm{lr}, k - 1, m) \wedge \mathrm{tk_le}(\mathrm{lr}, k, m) \implies \mathrm{mk_eq}(\mathrm{lr}, k, m))$
      NEW-CLAUSE$(\mathrm{mk_eq}(\mathrm{lr}, k - 1, m) \wedge \mathrm{tk_eq}(\mathrm{lr}, k, m + 1) \implies \mathrm{mk_eq}(\mathrm{lr}, k, m + 1))$
  if CHECK-SAT **then**
    Assert L DFA from the model proves the machine using Algorithm 5.
    return true
  else return false

7 Bouncers

Acknowledgement. Sincere thanks to Tony Guilfoyle who initially implemented a decider for bouncers[17]. Others have contributed to this method by producing alternative implementations (see Section 7.3) or discussing and writing the formal proof presented here: savask, Iijil, mei, Tristan Stérin (cosmo), Justin Blanchard, Pascal Michel.

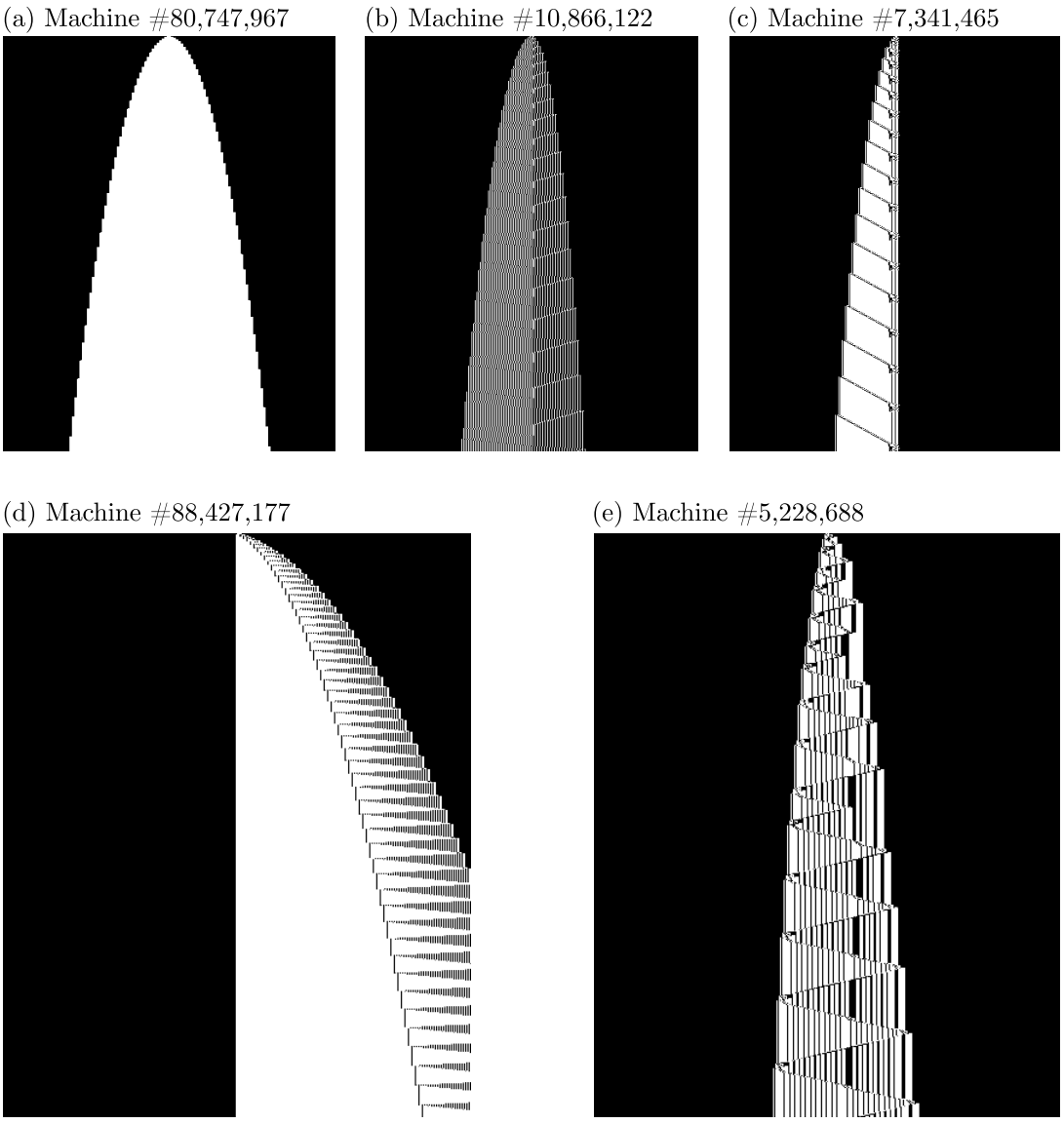


Figure 9: Space-time diagrams (10,000 steps) of several `bbchallenge` *bouncers*: (a) simple bouncer bouncing back and forth between expanding tape extremeties while writing 1s (b) bouncer with more complex alternating *repeater* patterns left and right of the origin (c) unilateral bouncer with a complex *wall* pattern at the origin (d) unilateral bouncer, main example used throughout this section (e) bouncer entering a repetitive bouncing pattern after ∼6,000 steps (bottom half of the image).

7.1 Characterising bouncers

Intuitively, a *bouncer* is a Turing machine that populates a tape with linearly-expanding patterns, called *repeaters*, possibly separated or enclosed by fixed patterns called *walls*. This intuitive definition corresponds to a wide range of behaviors, from simply bouncing back and forth between the tape's expanding extremities, Figure 9 (a), to complex traversal of *repeater* and *wall* patterns, Figure 9 (b)-

[17] See: `https://github.com/TonyGuil/bbchallenge/tree/main/Bouncers`.

(d), and possibly a delayed onset of the *bouncing* pattern, Figure 9 (e). What we call bouncers is a generalisation of the various classes of "Christmas trees" used to solve BB(4) [3].

The goal of this section is to formally characterise bouncers and show that they do not halt, see Theorem 7.9. Then, in Section 7.3, we show how to detect bouncers algorithmically in practice, see Algorithm 9.

7.1.1 Directional Turing machines

We build on the concept of directional Turing machine introduced in Section 1. Directional Turing machines are an equivalent formulation of Turing machines where the machine head lives in between the tape cells and can point to the left or to the right. We choose here to treat 0^∞ as a unique symbol (instead of an infinite collection of 0s) and write $\overline{\Sigma} = \{0^\infty\} \cup \Sigma$, with Σ the tape alphabet of the machine – in the context of BB(5), we have $\Sigma = \{0, 1\}$. For each Turing machine with set of states S we introduce $2|S|$ new configuration symbols (i.e. symbols used to describe machine configurations and not read/write symbols) denoting the machine head in two possible orientations, namely $\Delta = \{\overset{s}{\triangleleft} | s \in S\} \cup \{\overset{s}{\triangleright} | s \in S\}$.

We define a *tape*[18] to be a finite word of the form uhv, where $u, v \in \overline{\Sigma}^*$ and $h \in \Delta$, moreover, u and v must have at most one occurrence of 0^∞ each, respectively as first symbol for u or last symbol for v. We choose the initial tape to be $0^\infty \overset{A}{\triangleright} 0^\infty$. Now, we define tape rewrite rules which will be used to simulate a directional Turing machine which we fix from now on. Suppose that for $s \in S, x \in \Sigma$ we have $\delta(s, x) = (s', d, x')$ where $s' \in S, d \in \{\mathrm{L}, \mathrm{R}\}, x' \in \Sigma$ and δ the transition function of the machine, then we define the following tape rewrite rules:

If $d = \mathrm{L}$	If $d = \mathrm{R}$
$x \overset{s}{\triangleleft} \to \overset{s'}{\triangleleft} x'$	$x \overset{s}{\triangleleft} \to x' \overset{s'}{\triangleright}$
$\overset{s}{\triangleright} x \to \overset{s'}{\triangleleft} x'$	$\overset{s}{\triangleright} x \to x' \overset{s'}{\triangleright}$

If $x = 0$, also define:

$0^\infty \overset{s}{\triangleleft} \to 0^\infty \overset{s'}{\triangleleft} x'$	$0^\infty \overset{s}{\triangleleft} \to 0^\infty x' \overset{s'}{\triangleright}$
$\overset{s}{\triangleright} 0^\infty \to \overset{s'}{\triangleleft} x' 0^\infty$	$\overset{s}{\triangleright} 0^\infty \to x' \overset{s'}{\triangleright} 0^\infty$

Given a tape $t = uvw$ and a word $t' = uv'w$, with $u, w \in \overline{\Sigma}^*$, $v, v' \in (\overline{\Sigma} \cup \Delta)^*$, suppose that there is a rewrite rule $v \to v'$. Then t' is also a tape and in this situation we write $t \vdash t'$ meaning that t' is obtained from t by one simulation step; note that the definition is sound because, to any given tape, at most one rewrite rule applies, hence $v \to v'$ is defined uniquely and so is $t \vdash t'$. Note that the only case where $\vdash$ is not defined on a tape is if the machine halts in this configuration, i.e. an undefined transition is reached. Applying $\vdash$ successively $n \in \mathbb{N}$ times is written $\vdash^n$. The transitive closure of $\vdash$ (i.e. applying $\vdash$ one or more times) is written $\vdash^+$; applying $\vdash$ zero or more times is written $\vdash^*$.

Example 7.1. Consider `bbchallenge` machine[19] #88,427,177 (Figure 9 (d)), that has the following transition table:

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

[18] Note that in the terminology of Section 1, the definition of a tape that we use here, where the head in part of the tape, corresponds to a (partial) TM configuration.

[19] Accessible at `https://bbchallenge.org/88427177`

Given the above definitions, we will have the following rewrite rules with state A on the left-hand side:

$$\begin{aligned}
0 \overset{A}{\triangleleft} &\rightarrow 1 \overset{B}{\triangleright} \\
\overset{A}{\triangleright} 0 &\rightarrow 1 \overset{B}{\triangleright} \\
1 \overset{A}{\triangleleft} &\rightarrow \overset{E}{\triangleleft} 1 \\
\overset{A}{\triangleright} 1 &\rightarrow \overset{E}{\triangleleft} 1 \\
0^\infty \overset{A}{\triangleleft} &\rightarrow 0^\infty 1 \overset{B}{\triangleright} \\
\overset{A}{\triangleright} 0^\infty &\rightarrow 1 \overset{B}{\triangleright} 0^\infty
\end{aligned}$$

The other rewrite rules are those with left-hand side states B, C, D and E. Simulating the machine for 4 steps, starting from initial tape yields: $0^\infty \overset{A}{\triangleright} 0^\infty \vdash 0^\infty 1 \overset{B}{\triangleright} 0^\infty \vdash 0^\infty 1 \overset{C}{\triangleleft} 1 0^\infty \vdash 0^\infty 1 \overset{C}{\triangleright} 1 0^\infty \vdash 0^\infty 11 \overset{C}{\triangleright} 0^\infty$. Hence, $0^\infty \overset{A}{\triangleright} 0^\infty \vdash^* 0^\infty 11 \overset{C}{\triangleright} 0^\infty$.

7.1.2 Wall-repeater formula tapes

In this section, we formalise the above-stated intuition that bouncers expand a tape by repeating *repeater* patterns between fixed *walls*. Then, we formally define bouncers in Definition 7.8 and show that they do not halt in Theorem 7.9. For this, we are going to express bouncers' tapes using abbreviated regular expressions over the alphabet $\Delta \cup \overline{\Sigma}$. Given $u \in \Sigma^*$, we represent the regular language $\{u\}^*$ (zero or more repetitions of the word u), as (u). Also, we write $\Sigma^+ = \Sigma^* \setminus \{\varnothing\}$. We define a *wall-repeater formula tape* to be an expression of the form:

$$w_1(r_1)w_2(r_2)\dots w_n(r_n)w_{n+1}hw'_1(r'_1)w'_2(r'_2)\dots w'_m(r'_m)w'_{m+1} \tag{7.1}$$

if the following conditions are met:

$$\begin{aligned}
&n, m \geq 0, \\
&h \in \Delta, \\
&r_1, \dots, r_n, r'_1, \dots, r'_m \in \Sigma^+, \\
&w_2, \dots, w_{n+1}, w'_1, \dots, w'_m \in \Sigma^*, \\
&w_1, w'_{m+1} \in \overline{\Sigma}^*
\end{aligned}$$

and, as in the definition of a tape, w_1 (resp. w'_{m+1}) either does not contain the symbol 0^∞ or it starts with it (resp. ends with it). Words w_i, w'_j are called *walls* and can be empty, while the *repeaters*, r_i, r'_j must be nonempty. Note that we allow $n = m = 0$, hence a usual tape is also a wall-repeater formula tape. For the rest of this section, we abbreviate wall-repeater formula tapes as *formula tapes*.

Given a formula tape f let $\mathcal{L}(f)$ denote the language described by f, i.e. the set of tapes that match it.

Example 7.2. Consider the formula tape $f = 0 \overset{D}{\triangleright} (01)$. We have $\mathcal{L}(f) = \{0 \overset{D}{\triangleright}, 0 \overset{D}{\triangleright} 01, 0 \overset{D}{\triangleright} 0101, \dots\}$. Consider the formula tape $f' = 0^\infty(111)1110 \overset{A}{\triangleleft} 010101(01)10^\infty$. Then: $0^\infty 1110 \overset{A}{\triangleleft} 01010110^\infty$, $0^\infty 1111110 \overset{A}{\triangleleft} 01010110^\infty$, and $0^\infty 1110 \overset{A}{\triangleleft} 0101010110^\infty$ are elements of $\mathcal{L}(f')$.

Extending $\vdash$ to formula tapes: *shift rules*. We wish to extend the Turing machine step relation $\vdash$ (see Section 7.1.1) to formula tapes. This is quite straightforward when the head is pointing at a symbol of a wall (one of the w_i, w'_j in (7.1)): we simply apply a standard Turing machine step, leaving the definition of $\vdash$ unchanged.

However, we need to handle the case where the head is pointing at a repeater (one of the r_i, r'_j in (7.1)). Suppose that for some $u \in \Sigma^*$ and $r, \tilde{r} \in \Sigma^+$ and some state $s \in S$ we have $u \overset{s}{\triangleright} r \vdash^+ \tilde{r}u \overset{s}{\triangleright}$. Then, for any $n \geq 0$, we have $u \overset{s}{\triangleright} r^n \vdash^* \tilde{r}^n u \overset{s}{\triangleright}$. This motivates the definition of (right) *shift rules*, that is, rewrite rules for formula tapes, which rewrite a subword $u \overset{s}{\triangleright} (r)$ into $(\tilde{r})u \overset{s}{\triangleright}$, denoted by $u \overset{s}{\triangleright} (r) \rightarrow (\tilde{r})u \overset{s}{\triangleright}$. Similarly, left shift rules are of the form $(r) \overset{s}{\triangleleft} u \rightarrow \overset{s}{\triangleleft} u(\tilde{r})$ given that $r \overset{s}{\triangleleft} u \vdash^+ \overset{s}{\triangleleft} u\tilde{r}$. Note that repeaters r and $\tilde{r}$ of a shift rule necessarily have the same size.

Hence, we have two cases to consider for defining $\vdash$ on the following formula tape f (as defined in (7.1)):

$$f = w_1(r_1)w_2(r_2)\dots w_n(r_n)w_{n+1}hw'_1(r'_1)w'_2(r'_2)\dots w'_m(r'_m)w'_{m+1}$$

1. (Usual step) If h points at nonempty w_{n+1} or w'_1, and $w_{n+1}hw'_1 \vdash \tilde{w}_{n+1}\tilde{h}\tilde{w}'_1$ as tapes, replace the terms $w_{n+1}hw'_1(r'_1)$ of f with $\tilde{w}_{n+1}\tilde{h}\tilde{w}'_1$. Call the new formula f', and define $f \vdash f'$. As for tapes, $\vdash$ is undefined if $w_{n+1}hw'_1$ corresponds to a halting configuration (i.e. undefined transition).

2. (Shift rule) Two cases:

   (a) Right shift rule. If $h = \overset{s}{\triangleright}$ with $s \in S$ and w'_1 is empty, consider the set of shift rules $\mathcal{R} = \{u \overset{s}{\triangleright} (r'_1) \to (\tilde{r})u \overset{s}{\triangleright} \mid \tilde{r} \in \Sigma^+,\ u \in \Sigma^* \text{ is a suffix of } w_{n+1}\}$. If $\mathcal{R}$ is not empty then apply the right shift rule of $\mathcal{R}$ with smallest u (possibly empty), call the new formula f', and we define $f \vdash f'$. If $\mathcal{R}$ is empty, $\vdash$ cannot be applied to f.

   (b) Left shift rule. If $h = \overset{s}{\triangleleft}$ with $s \in S$ and w_{n+1} is empty, consider the set of shift rules $\mathcal{R} = \{(r_n) \overset{s}{\triangleleft} u \to \overset{s}{\triangleleft} u(\tilde{r}) \mid \tilde{r} \in \Sigma^+,\ u \in \Sigma^* \text{ is a prefix of } w'_1\}$. If $\mathcal{R}$ is not empty then apply the left shift rule of $\mathcal{R}$ with smallest u (possibly empty), call the new formula f', and we have $f \vdash f'$. If $\mathcal{R}$ is empty, $\vdash$ cannot be applied to f.

From the above definition of $\vdash$ on a formula tape f, it is clear that (i) for usual tapes our new definition of $\vdash$ coincides with the old one, (ii) there is at most one formula tape f' such that $f \vdash f'$, and (iii) the only cases where $\vdash$ is not defined on a formula tape are when the machine halts (usual step case) or no shift rule applies (shift rule case). Moreover, we get the following result:

Lemma 7.3. Let f and f' be formula tapes with $f \vdash f'$. Then, for all $t \in \mathcal{L}(f)$, there exists some $t' \in \mathcal{L}(f')$ such that $t \vdash^* t'$ and, in case $f \vdash f'$ via usual step, $t \vdash^+ t'$.

Proof. If f' follows from f by a usual step, then applying one usual step to t yields $t' \in \mathcal{L}(f')$. If f' follows from f by a shift rule (of the form $u \overset{s}{\triangleright} (r) \vdash^k (\tilde{r})u \overset{s}{\triangleright}$ or $(r) \overset{s}{\triangleleft} u \vdash^k \overset{s}{\triangleleft} u(\tilde{r})$), and the corresponding repeater (r) is used n times in the match $t\ \in \mathcal{L}(f)$, we apply nk steps to t to obtain $t' \in \mathcal{L}(f')$. This amounts to a positive number of steps (justifying $\vdash^+$) except in case of a shift rule applied $k = 0$ times. □

Example 7.4. Taking the machine of Example 7.1, we have the right shift rule $0 \overset{D}{\triangleright} (01) \to (11)0 \overset{D}{\triangleright}$. Indeed, this is because: $0 \overset{D}{\triangleright} 01 \vdash 0 \overset{A}{\triangleleft} 11 \vdash 1 \overset{B}{\triangleright} 11 \vdash 11 \overset{D}{\triangleright} 1 \vdash 110 \overset{D}{\triangleright}$, hence $0 \overset{D}{\triangleright} 01 \vdash^* 110 \overset{D}{\triangleright}$, giving the shift rule. Consider the tape formula of previous Example 7.2: $f' = 0^\infty(111)1110 \overset{A}{\triangleleft} 010101(01)10^\infty$. We have $f' \vdash^{13} 0^\infty(111)1111110110 \overset{D}{\triangleright} (01)10^\infty$, at this point the head points at a repeater and the set of applicable right shift rules is $\mathcal{R} = \{0 \overset{D}{\triangleright} (01) \to (11)0 \overset{D}{\triangleright},\ 10 \overset{D}{\triangleright} (01) \to (11)10 \overset{D}{\triangleright},\ 110 \overset{D}{\triangleright} (01) \to (11)110 \overset{D}{\triangleright}\}$, and following the definition of $\vdash$, we apply $0 \overset{D}{\triangleright} (01) \to (11)0 \overset{D}{\triangleright}$ as it has the smallest left-hand side, giving: $0^\infty(111)111111011(11)0 \overset{D}{\triangleright} 10^\infty$.

Aligning formula tapes. One last tool that we need before characterising bouncers and proving that they do not halt (Theorem 7.9) is formula tape *alignment* (Definition 7.5): sometimes it is necessary to rewrite a formula tape in an equivalent, *aligned* form in order for any shift rules to apply.

Definition 7.5 (Alignment operator)**.** Take a formula tape, as given in (7.1):

$$f = w_1(r_1)w_2(r_2)\dots w_n(r_n)w_{n+1}hw'_1(r'_1)w'_2(r'_2)\dots w'_m(r'_m)w'_{m+1}$$

The alignment operator $f \mapsto \mathcal{A}(f)$ moves repeaters away from the head h by repeatedly applying any of the following rules until none apply anymore:

1. Replace $(r'_j)v$ with $v(r)$ in f, if $r'_j v = vr$ with v a nonempty prefix of w'_{j+1}, $r \in \Sigma^+$, and $1 \le j \le m$.

2. Replace $v(r_i)$ with $(r)v$ in f, if $vr_i = rv$ with v a nonempty suffix of w_i, $r \in \Sigma^+$, and $1 \le i \le n$.

Clearly, the order application of these rules does not matter, i.e. $\mathcal{A}$ is well-defined, and $\mathcal{A}(\mathcal{A}(f)) = \mathcal{A}(f)$.

Lemma 7.6. For any formula tape f, $\mathcal{L}(\mathcal{A}(f)) = \mathcal{L}(f)$, i.e. both f and $\mathcal{A}(f)$ represent the same set of tapes.

Proof. Consider an alignment rule as in case (1) of Definition 7.5: replacing $(r'_j)v$ with $v(r)$ if $r'_j v = vr$. Then, for all $n \in \mathbb{N}$, ${r'_j}^n v = vr^n$, hence $(r'_j)v$ and $v(r)$ describe the same language: $\mathcal{L}((r'_j)v) = \mathcal{L}(v(r))$. Same for case (2) and for multiple applications of (1) and (2) in any order, hence we have $\mathcal{L}(\mathcal{A}(f)) = \mathcal{L}(f)$. □

Example 7.7. Take $f = 0^\infty(111)1111 \overset{\text{B}}{\triangleright} (01)01010110^\infty$ for the machine given in Example 7.1. One can verify that no right shift rule applies to f hence $\vdash$ does not apply to f. However, we have $\mathcal{A}(f) = 0^\infty(111)1111 \overset{B}{\triangleright} 010101(01)10^\infty$, $\mathcal{L}(\mathcal{A}(f)) = \mathcal{L}(f)$, and we can apply $\vdash$ to $\mathcal{A}(f)$ by performing usual steps: $\mathcal{A}(f) \vdash^{12} 0^\infty(111)1111110110 \overset{D}{\triangleright} (01)10^\infty$ and now the shift rule of Example 7.4 can apply, giving $0^\infty(111)111111011(11)0 \overset{\text{D}}{\triangleright} 10^\infty$. Another alignment example is $f = 0^\infty101(11)0 \overset{\text{D}}{\triangleright} 10(100)110^\infty$ for which $\mathcal{A}(f) = 0^\infty10(11)10 \overset{\text{D}}{\triangleright} 101(001)10^\infty$.

The above example shows that alignment (which preserves the set of recognised tapes) can allow to run a shift rule on f' with $\mathcal{A}(f) \vdash^* f'$ when no shift rule was applicable directly to f. In fact, one can show that the alignment operator can only *increase* the number of applicable shift rules: if a shift rule is applicable to f then a shift rule applicable to f' with $\mathcal{A}(f) \vdash^* f'$ can always be found. This motivates the introduction of the notation $f \vdash_{\mathcal{A}} f'$ which means that we have $\mathcal{A}(f) \vdash f'$.

Given two formula tapes f and f' we will say that f' is a *special case* of f, if $\mathcal{A}(f')$ can be obtained from $\mathcal{A}(f)$ by replacing subwords of the form (r) in $\mathcal{A}(f)$ by $r^n(r)r^m$ for some $n, m \geq 0$ and $r \in \Sigma^+$. Note that because of alignment, $n = 0$ (resp. $m = 0$) if (r) in $\mathcal{A}(f)$ is to the left (resp. to the right) of the head. If f' is a special case of f then $\mathcal{L}(f') \subseteq \mathcal{L}(f)$, and the authors conjecture that the converse is true under some mild additional assumptions[20].

We finally get to the main result of this section, which characterises bouncers formally – a bouncer is any machine to which Theorem 7.9 applies:

Definition 7.8 (Bouncers)**.** A bouncer is a Turing machine such that there exists a wall-repeater formula tape f which satisfies:

1. There is a reachable tape $t \in \mathcal{L}(f)$, i.e. $0^\infty \overset{\text{A}}{\triangleright} 0^\infty \vdash^* t$
2. There is a formula tape f' such that $f \vdash^+_{\mathcal{A}} f'$ and f' is a special case of f

We say that formula tape f *solves* the bouncer.

Theorem 7.9. A bouncer does not halt.

Proof. Unraveling $f \vdash^+_{\mathcal{A}} f'$ gives $n > 0$ and $f = f_0, \ldots, f_n$ with $f_n = f'$ and $\mathcal{A}(f_i) \vdash f_{i+1}$ for $0 \leq i < n$. It follows from Lemma 7.3 and Lemma 7.6 that there exist tapes $t_0, \ldots, t_n$, such that $t_0 = t$ and $t_i \in \mathcal{L}(f_i)$ for $0 \leq i \leq n$, and $t_i \vdash^* t_{i+1}$ for $0 \leq i < n$, giving $t_0 \vdash^* t_n$. Moreover, $t_0 \vdash^+ t_n$ if the any of the $\mathcal{A}(f_i) \vdash f_{i+1}$ relations are via usual steps.

Indeed, this must be the case. Suppose instead we had a sequence of shift rules. Each one would preserve the direction of the head, and increment the number of repeaters behind the head in the formula tape. However, these properties are unchanged by passing from a formula tape f to $\mathcal{A}(f)$ or a special case of f. It would follow that f_0 has strictly more repeaters behind the head than f_0, a contradiction.

Since f_n is a special case of f_0, we have $\mathcal{L}(\mathcal{A}(f_n)) \subseteq \mathcal{L}(\mathcal{A}(f_0))$, and using Lemma 7.6, we have $\mathcal{L}(f_n) \subseteq \mathcal{L}(f_0)$ and thus, $t_n \in \mathcal{L}(f_0)$. We can repeat this construction indefinitely and yield an infinite sequence of tapes $(t_n)_{n\in\mathbb{N}}$ such that $t \vdash^+ t_i$ for all $i \in \mathbb{N}$, hence the machine does not halt. □

Definition 7.10 (Bouncer certificate)**.** For a given bouncer, a bouncer certificate consists at least of the following:

1. The formula tape f of Definition 7.8 that solves the bouncer.
2. The time step at which tape t such that $t \in \mathcal{L}(f_0)$ is reached from $0^\infty \overset{\text{A}}{\triangleright} 0^\infty$.
3. The number of *macro steps* n such that $f \vdash^n_{\mathcal{A}} f'$ with f' special case of f, where a macro step on a formula tape consists in applying $\vdash_{\mathcal{A}}$, i.e. alignment followed by $\vdash$ (one usual step or one shift rule step).

Given such certificate, one can verify that the machine is a bouncer by applying Theorem 7.9 or that the certificate is erroneous. Additional information, such has the list of used shift rules can be added to the certificate for convenience.

[20] See `https://discuss.bbchallenge.org/t/186`.

Example 7.11. The machine of Example 7.1 (Figure 9 (d)) that is used in our series of examples for this section, is a bouncer. Indeed, we have $0^\infty \overset{A}{\triangleright} 0^\infty \vdash^{64} 0^\infty 11111101100 \overset{D}{\triangleright} 0^\infty$. The tape $t = 0^\infty \vdash^{64} 0^\infty 11111101100 \overset{D}{\triangleright} 0^\infty$ is in the language the following formula tape:

$$f_0 = 0^\infty(111)1110(11)00 \overset{D}{\triangleright} 0^\infty$$

At this point, and for the next 25 usual steps alignment does not affect the formulas, and we get:

$$f_{25} = 0^\infty(111)1110(11) \overset{A}{\triangleleft} 01010110^\infty$$

One shift rule gives:

$$f_{26} = 0^\infty(111)1110 \overset{A}{\triangleleft} (01)01010110^\infty$$

After alignment:

$$\mathcal{A}(f_{26}) = 0^\infty(111)1110 \overset{A}{\triangleleft} 010101(01)10^\infty$$

From there, $\mathcal{A}(f_{26}) \vdash f_{27}$ with:

$$f_{27} = 0^\infty(111)1111 \overset{B}{\triangleright} 010101(01)10^\infty$$

After 12 usual steps, not affected by alignment, we arrive at:

$$f_{39} = 0^\infty(111)1111110110 \overset{D}{\triangleright} (01)10^\infty$$

One shift rule gives:

$$f_{40} = 0^\infty(111)111111011(11)0 \overset{D}{\triangleright} 10^\infty$$

Aligning gives:

$$\mathcal{A}(f_{40}) = 0^\infty(111)1111110(11)110 \overset{D}{\triangleright} 10^\infty$$

Finally, from there $\mathcal{A}(f_{40}) \vdash f_{41}$ with one usual step:

$$f_{41} = \mathcal{A}(f_{41}) = 0^\infty(111)1111110(11)1100 \overset{D}{\triangleright} 0^\infty$$

Now, f_{41} is a special case of f_0 because $\mathcal{A}(f_{41})$ differs from $f_0 = \mathcal{A}(f_0)$ only by including one repetition of repeaters (111) and (11) in the walls directly to their right. The assumptions of Theorem 7.9 hold and our Turing machine is a bouncer: it does not halt.

A bouncer certificate (Definition 7.10) for this machine is $f_0 = 0^\infty(111)1110(11)00 \overset{D}{\triangleright} 0^\infty$, time step 64 at which $0^\infty 11111101100 \overset{D}{\triangleright} 0^\infty \in \mathcal{L}(f_0)$ is reached and 41 macro steps which transform f_0 into special case f_{41} under successive $\vdash$ and alignement applications.

Note that Cyclers (Section 2) and Translated Cyclers (Section 3) are special cases of bouncers, as they satisfy Theorem 7.9.

7.1.3 Linear-quadratic growth

For a given formula tape, using the notation of Equation 7.1:

$$f = w_1(r_1)w_2(r_2)\dots w_n(r_n)w_{n+1}hw'_1(r'_1)w'_2(r'_2)\dots w'_m(r'_m)w'_{m+1}$$

call $C_f(k) \in \mathcal{L}(f)$ the tape where each repeater is used exactly $k \in \mathbb{N}$ times:

$$C_f(k) = w_1 r_1^k w_2 r_2^k \dots w_n r_n^k w_{n+1} h w'_1 {r'_1}^k w'_2 {r'_2}^k \dots w'_m {r'_m}^k w'_{m+1}$$

We denote an infinite sequence $u_0, u_1, \dots u_n, \dots$ with $n \in \mathbb{N}$ by $(u_n)_{n\in\mathbb{N}}$. We define the difference operator $D : \mathbb{Z}^\mathbb{N} \to \mathbb{Z}^\mathbb{N}$ via $D((u_n)_{n\in\mathbb{N}}) = (u_{n+1} - u_n)_{n\in\mathbb{N}}$. The *length* of a tape t is the number of symbols of Σ in t, i.e. ignoring head and 0^∞.

Theorem 7.9 characterises bouncers but we lack a practical criterion for recognising a bouncer from its sequence of successive tapes starting from $0^\infty \overset{A}{\triangleright} 0^\infty$. We make a step in that direction by showing that if a bouncer solved by formula tape f that is not a cycler (Section 2), then we can construct from f a new formula tape $\text{sync}(f)$, that also solves the bouncer but such that tapes $C_{\text{sync}(f)}(k)$, where all repeaters are synchronously repeated k times, are reached by the machine for all successive $k \in \mathbb{N}$. Importantly for being detected in practice, the length of tapes $C_{\text{sync}(f)}(k)$ grows linearly in quadratic time:

Theorem 7.12 (Linear-quadratic growth)**.** Let M be a bouncer that is not a cycler, solved by some formula tape f. Call $(t_n)_{n\in\mathbb{N}}$ the sequence of tapes that it visits starting from $t_0 = 0^\infty \overset{\mathrm{A}}{\triangleright} 0^\infty$. Call l_n the length of t_n. Then, there is a formula tape called $\mathrm{sync}(f)$, obtained from f, that also solves the bouncer such that there is an extraction function $g : \mathbb{N} \to \mathbb{N}$ with $g(n) \geq n$ for all $n \in \mathbb{N}$, satisfying:

1. For all $n \in \mathbb{N}$, $t_{g(n)} = C_{\mathrm{sync}(f)}(n)$.
2. $(l_{g(n)})_{n\in\mathbb{N}}$ is an arithmetic progression, i.e. $D((l_{g(n)})_{n\in\mathbb{N}}) = (K)_{n\in\mathbb{N}}$ for some constant $K \in \mathbb{N}$.
3. $(g(n))_{n\in\mathbb{N}}$ is a quadratic progression, i.e. $D(D((g(n))_{n\in\mathbb{N}})) = (K')_{n\in\mathbb{N}}$ for some constant $K' \in \mathbb{N}$.

Proof. For simplicity of notations, we suppose that there are no repeaters after the tape head in f, as we would apply exactly the same transformations after the head. Using Equation (7.1), we write $f = w_1(r_1)w_2(r_2)\dots w_n(r_n)w_{n+1}hw$ with $n \geq 1$, $w_i \in \Sigma^*$ for $2 \leq i \leq n+1$, $w_1, w \in \overline{\Sigma}^*$ and $r_i \in \Sigma^+$ for $1 \leq i \leq n$ and $h \in \Delta$. Without loss of generality we suppose that f is aligned (otherwise we just apply $\mathcal{A}$ to it).

Because f solves the bouncer, by Definition 7.8, the machine reaches tape t of the form $t = w_1 r_1^{k_1} w_2 r_2^{k_2} \dots r_n^{k_n} w_{n+1} hw$ with $k_1, \dots, k_n \in \mathbb{N}$. By Theorem 7.9, we have $t \vdash^+ t'$ with $t' \in \mathcal{L}(f')$ and $t' = w_1 r_1^{k_1+p_1} w_2 r_2^{k_2+p_2} \dots r_n^{k_n+p_n} w_{n+1} hw$ with $p_1, \dots, p_n \in \mathbb{N}$ and same w_i and r_i as f because f is aligned. Note that p_i is positive because shift rules preserve the size of repeaters. Because $t' \in \mathcal{L}(f') \subseteq \mathcal{L}(f)$ we can repeat this indefinitely and get $t \vdash^+ w_1 r_1^{k_1+Np_1} w_2 r_2^{k_2+Np_2} \dots r_n^{k_n+Np_n} w_{n+1} hw$ for all $N \in \mathbb{N}$.

Hence, define $\mathrm{sync}(f) = w_1 r_1^{k_1}(r^{p_1}) w_2 r_2^{k_2}(r^{p_2}) \dots w_n r_n^{k_n}(r^{p_n}) w_{n+1} hw$. When $p_i = 0$, then $r^{p_i} = \varnothing$ and we discard it. All p_i cannot be 0, otherwise the machine is a cycler which is excluded. Altogether, we can write: $\mathrm{sync}(f) = \tilde{w}_1(\tilde{r}_1)\tilde{w}_2(\tilde{r}_2)\dots\tilde{w}_m(\tilde{r}_m)\tilde{w}_{m+1}hw$ with $2 \leq m \leq n$ and nonempty $\tilde{r}_i$, which is a valid formula tape which also solves the bouncer. By construction, the machine will reach tapes of the form $C_{\tilde{f}}(n)$ for all $n \in \mathbb{N}$ at increasing time steps. We immediately get Point 2 because $C_{\tilde{f}}(n+1)$ contains $K = \sum_{i=1}^{n} |\tilde{r}_i|$ more symbols than $C_{\tilde{f}}(n)$, which is a constant.

Concerning Point 3, call $g(n)$ the time step at which $C_{\tilde{f}}(n)$ is reached. By construction, $g(n) \geq n$. The number of macro steps $M \geq 1$ such that we get $\mathrm{sync}(f) \vdash_{\mathcal{A}}^{M} f''$ with f'' special case of $\mathrm{sync}(f)$ is a constant. We write $M = M_u + M_s$ with M_u the number of usual steps and M_s the number of shift rule applications. Call M'_s the number of Turing machine steps taken in total to perform the base case (i.e. the steps needed to prove that a shift rule is correct) of each of the M_s shift rules. When processing $C_{\tilde{f}}(n)$, each shift rule is applied once each to the n repetitions of each repeater (because shift rules preserve the number of repetitions), hence we get: $g(n+1) - g(n) = M_u + nM'_s$, which means $D(D((g(n))_{n\in\mathbb{N}})) = (M'_s)_{n\in\mathbb{N}}$ which is constant, as needed.

□

Example 7.13. Using Example 7.11, we get that $f = 0^\infty(111)1110(11)00 \overset{\mathrm{D}}{\triangleright} 0^\infty$ solves the bouncer given in Example 7.1. Using Theorem 7.12, we have $\mathrm{sync}(f) = 0^\infty(111)1111110(11)1100 \overset{\mathrm{D}}{\triangleright} 0^\infty$. We have $\mathrm{sync}(f) \vdash_{\mathcal{A}}^{M} f'$ with $M = 47$ and f' special case of $\mathrm{sync}(f)$. Note that $47 > 41$ found in Example 7.11 because $\mathrm{sync}(f)$ is a bit bigger than f.

We can decompose $M = M_u + M_s$ with $M_u = 45$ usual steps and $M_s = 2$ shift rule applications. The shift rules that are used are $0 \overset{\mathrm{D}}{\triangleright} (01) \to (11)0 \overset{\mathrm{D}}{\triangleright}$ which takes 4 Turing machine steps to execute (Example 7.4) and $(11) \overset{\mathrm{A}}{\triangleleft} \to \overset{\mathrm{A}}{\triangleleft} (01)$ which takes 2, hence $M'_s = 6$.

Calling $g(n)$ the time step at which $C_{\mathrm{sync}(f)}(n)$ is reached, we have $g(0) = 64$. Using the proof of Theorem 7.12, $g(n+1) - g(n) = M_u + nM'_s = 45 + 6n$, hence, $g(n) = 3n^2 + 42n + 64$. We also have $l_{g(n)} = 11 + 5n$. One can check by simulation that tapes $C_{\mathrm{sync}(f)}(n)$ are reached at time steps $g(n)$.

7.2 Deciding bouncers in practice

In this section, we use the theory presented above in order to give a practical and efficient algorithm for deciding bouncers in practice.

7.2.1 Formula tape fitting

We introduce $\mathcal{A}_r$, the *right-alignment* operator on formula tapes which only applies rule of type 1 in Definition 7.5 to all repeaters, both before and after the formula's head, bubbling them to the right.

In this section, we give a greedy algorithm (Algorithm 8) that fits a formula tape f only given three tapes: $C_f(0)$, $C_f(1)$ and $C_f(2)$. The algorithm is guaranteed to work on right-aligned formula tapes with nonempty *intermediary walls*, which are all walls but the first and last one, see Theorem 7.18. First, we show that the precondition of having nonempty intermediary walls is without loss of generality, i.e. that all bouncers can be solved using such formulas. We prove this result in Lemma 7.16, using two technical lemmas, Lemmas 7.14 and 7.15.

Lemma 7.14. Assume that M is a bouncer solved by some formula tape f, then:

1. $\mathcal{A}_r(f)$ also solves the bouncer.
2. Any special case f' of f also solves the bouncer.

Proof. 1. We have $\mathcal{A}(\mathcal{A}_r(f)) = \mathcal{A}(f)$ hence, Theorem 7.9 will reach $\mathcal{A}(f)$ after one application of $\mathcal{A}$ in any case and the bouncer will get solved in the same way.

2. By definition of special case f' can be constructed from f by replacing any (r) by $r^n(r)r^m$ for some $n, m \in \mathbb{N}$. Aligning f' will have the same effect as aligning f with the addition that repeaters before the head will look like $(r)r^{n+m}$ and after the head $r^{n+m}(r)$. Then, shift rules will apply similarly to f' and f, with the addition in f' that segments of the form r^{n+m} will be handled by simulating the same shift rule as for (r) through usual steps. Hence, f' solves the bouncer.

□

Lemma 7.15. Let $a, b \in \Sigma^+$ be two nonempty words such that we have $a[i \bmod |a|] = b^\infty[i]$ for all $i \in \mathbb{N}$, with $b^\infty[i]$ being the character at position i in the infinite concatenation of word b with itself. Call $k = \gcd(|a|, |b|)$. Then, a and b are powers of the same word c such that $a = c^m$ and $b = c^n$, with c the first k characters of a and b, and $m = |a|/k$ and $n = |b|/k$.

Proof. Note that $|a| \neq 0$ and $|b| \neq 0$ by hypothesis. Call $f(i) = b^\infty[i]$. It is immediate that $|b|$ is a period of f. By hypothesis, we also have that $|a|$ is a period of f. Hence, any linear combination that has positive value of $|a|$ and $|b|$ is also a period of f. Hence, by Euclid's algorithm, $k = \gcd(|a|, |b|)$ is a period of f, from which we get $b = c^{|b|/k}$ with c the first k characters of b since $k \leq |b|$. By hypothesis, we know that $a^\infty = b^\infty$, hence, we have $a = c^{|a|/k}$, as needed.

□

Lemma 7.16. Let a machine M be a bouncer solved by formula tape f, then f can be transformed into a formula tape f' such that $\mathcal{A}_r(f')$ has nonempty intermediary walls that solves the bouncer.

Proof. By Lemma 7.14, $\mathcal{A}_r(f)$ also solves the bouncer. For each case $\dots(r_1)(r_2)\dots$ where two repeaters in $\mathcal{A}_r(f)$ are separated by an empty wall, apply the following transformation to $\mathcal{A}_r(f)$, according to two cases:

1. There is $k \geq 1$ such that right-aligning $(r_1)r_2^k(r_2)$ yields a non-empty intermediary wall, replace $(r_1)(r_2)$ by $\mathcal{A}_r((r_1)r_2^k(r_2))$, yielding a new formula tape f'. Noticing that $(r_1)r_2^k(r_2)$ is a special case of $(r_1)(r_2)$ and combining both points of Lemma 7.14 we get that f' still solves the bouncer.
2. For all $k \geq 1$ there is $r'_k \in \Sigma^+$ such that we have $\mathcal{A}_r((r_1)r_2^k(r_2)) = r_2^k(r'_k)(r_2)$. We deduce that $r_1[i \bmod |r_1|] = r_2^\infty[i]$ for all $i \in \mathbb{N}$. By Lemma 7.15 we get that there is c such that $r_1 = c^m$ and $r_2 = c^n$. Hence, we can replace $(r_1)(r_2) = (c^m)(c^n)$ by $(c^m c^n) = (r_1 r_2)$ in the formula tape and still solve the bouncer with this new formula tape. Indeed, because f solves the bouncer we have $u \in \Sigma^*$, $s \in S$, $r', r'' \in \Sigma^+$ and two shift rules, for instance, right shift rules: $u \overset{s}{\triangleright} (c^m) \to (r')u \overset{s}{\triangleright}$ and $u \overset{s}{\triangleright} (c^n) \to (r'')u \overset{s}{\triangleright}$. Hence, considering $u \overset{s}{\triangleright} c^m c^n$, we get $u \overset{s}{\triangleright} c^m c^n \vdash^* r'u \overset{s}{\triangleright} c^n \vdash^+ r'r''u \overset{s}{\triangleright}$, meaning that we have a right shift rule $u \overset{s}{\triangleright} (c^m c^n) \to (r'r'')u \overset{s}{\triangleright}$, as needed.

By applying case (1) or (2) to all repeaters of $\mathcal{A}_r(f)$ that are separated by an empty wall, we get f', a transformation of f with nonempty intermediary walls that solves the bouncer. Because f' is right-aligned by construction, $\mathcal{A}_r(f') = f'$ and we get the result.

□

We are now ready to infer such a formula tape f from three examples $t_i = C_f(i)$. To simplify the procedure, we wish to drop the head and 0^∞ symbols from the tapes and re-attach them to the matching formula.

Given a tape t, we call $\mathcal{S}(t) \in \Sigma^*$ the *headless* version of t which is the tape without head and 0^∞ symbols. We also introduce headless formula tapes: $\mathcal{S}(f)$ is f without head and 0^∞ symbols.

Algorithm 8 is a greedy algorithm which fits a headless formula tape given three headless tapes t_0, t_1, t_2. It proceeds by recursively constructing the leftmost wall using the longest common prefix of t_0, t_1 and t_2 and then fits the leftmost repeater using the longest prefix r of t_1 such that rr is a prefix of t_2, we have:

Algorithm 8 Greedy formula tape fitting algorithm FitFormulaTape

```
1:  procedure HeadlessFormulaTape FitFormulaTape(Word t0, Word t1, Word t2)
2:      if t0.empty() && t1.empty() && t2.empty() then
3:          return HeadlessFormulaTape::empty()
4:
5:      if t0.len() > 0 && t1.len() > 0 && t2.len() > 0 && t0[0] == t1[0] && t1[0] == t2[0] then
6:          return HeadlessFormulaTape::symbol(t0[0]).concat(FitFormulaTape(t0[1:], t1[1:],
    t2[1:]))
7:
8:      uint longest_prefix_size = get_longest_prefix_size(t1, t2)
9:
10:     for l = longest_prefix_size; k ≥ 1; k -= 1 do
11:         if 2*l < t2.len() && t2[:l] == t2[l:2*l] then
12:             return HeadlessFormulaTape::repeater(t2[:l]).concat(FitFormulaTape(t0, t1[l:],
    t2[2*l:]))
13:
14:     raise Failure
```

Remark 7.17 (Implementation details of Algorithm 8)**.** We assume that we are given a **HeadlessFormulaTape** construct, and a function **get_longest_match_size** which returns the size of the longest prefix of two words. Furthermore, for an array `a`, we use the notation `a[b:e]` to mean the slice of this array between indices `b` and `e`, excluding `e`.

Theorem 7.18 (Greedy formula tape fitting)**.** Let f be a headless formula tape with nonempty intermediary walls such that $\mathcal{A}_r(f) = w_1(r_1)w_2(r_2)\dots w_m(r_m)w$ with $m \in \mathbb{N}$, $r_i, w_i \in \Sigma^+$ with $2 \leq i \leq m$, and $w_1, w \in \Sigma^*$. Take $t_0, t_1, t_2 \in \Sigma^+$ satisfying:

$$\begin{aligned} t_0 &= \mathcal{S}(C_f(0)) = w_1\ w_2 \dots w_m\ w \\ t_1 &= \mathcal{S}(C_f(1)) = w_1\ r_1\ w_2\ \dots w_m\ r_m\ w \\ t_2 &= \mathcal{S}(C_f(2)) = w_1\ r_1 r_1\ w_2\ \dots w_m\ r_m r_m\ w \end{aligned}$$

Then Algorithm 8, FitFormulaTape(t_0, t_1, t_2) does not raise failure and returns $\mathcal{A}_r(f)$.

Proof. Notation: if a word $u \in \Sigma^*$ is not empty we use the notation $u[0]$ to mean the first symbol of u. Let's first prove the case $m = 1$ and write $r = r_1$. Using the case of Algorithm 8, line 10, we get:

$$\begin{aligned} t_0 &= w_1\ w & & & t_0 &= w \\ t_1 &= w_1\ r\ w & &\rightarrow_{\text{Algorithm 8 goes to}} & t_1 &= r\ w \\ t_2 &= w_1\ rr\ w & & & t_2 &= rr\ w \\ f_{\text{out}} &= \varnothing & & & f_{\text{out}} &= w_1 \end{aligned}$$

Indeed, this is because w_1 is the biggest prefix shared by t_0, t_1 and t_2; otherwise, it means that $w_1 w[0]$ is a prefix of t_1 and we get $r[0] = w[0]$ which is excluded by right-alignment.

From there, we have two cases: (1) if $w = \varnothing$ then the greedy algorithm returns $f_{\text{out}} = w_1(r) = \mathcal{A}_r(f)$, as needed, (2) if $w \neq \varnothing$, because $\mathcal{A}_r(f)$ is right-aligned, we must have $w[0] \neq r[0]$ (r is not empty by hypothesis), and r is the longest prefix common to t_1 and t_2 since otherwise, using t_2 we get $w[0] = r[0]$. Hence, the greedy algorithm returns $f_{\text{out}} = w_1(r)w = \mathcal{A}_r(f)$, as needed.

Assuming $m \geq 2$, as above because of right-alignment, we get that w_1 is the longest prefix common to t_0, t_1 and t_2, hence:

$$\begin{aligned} t_0 &= w_1\ w_2\ \ldots w_m\ w \\ t_1 &= w_1\ r_1\ w_2\ r_2\ \ldots w_m\ r_m\ w \\ t_2 &= w_1\ r_1 r_1\ w_2\ r_2 r_2\ \ldots w_m\ r_m r_m\ w \\ f_{\text{out}} &= \varnothing \end{aligned} \quad \rightarrow_{\text{Algorithm 8 goes to}} \quad \begin{aligned} t_0 &= w_2\ \ldots w_m\ w \\ t_1 &= r_1\ w_2\ r_2\ \ldots w_m\ r_m\ w \\ t_2 &= r_1 r_1\ w_2\ r_2 r_2\ \ldots w_m\ r_m r_m\ w \\ f_{\text{out}} &= w_1 \end{aligned}$$

By hypothesis, w_2 and r_1 are not empty and, because $\mathcal{A}_r(f)$ is right-aligned, we get $w_2[0] \neq r_1[0]$. Hence, Algorithm 8 enters the repeater-fitting case on line 8. Necessarily, r_1 is the longest prefix to t_1 and t_2 otherwise using t_2 we get $r_1[0] = w_2[0]$ which contradicts right-alignment, hence:

$$\begin{aligned} t_0 &= w_2\ \ldots w_m\ w \\ t_1 &= w_2\ r_2\ \ldots w_m\ r_m\ w \\ t_2 &= w_2\ r_2 r_2\ \ldots w_m\ r_m r_m\ w \\ f_{\text{out}} &= w_1(r_1) \end{aligned}$$

From here we can inductively conclude that Algorithm 8 will reach:

$$\begin{aligned} t_0 &= w_m\ w \\ t_1 &= w_m\ r_m\ w \\ t_2 &= w_m\ r_m r_m\ w \\ f_{\text{out}} &= w_1(r_1) \ldots w_{m-1}(r_{m-1}) \end{aligned}$$

Using the same argument as for $m = 1$, we get that Algorithm 8 returns $f_{\text{out}} = w_1(r_1) \ldots w_m(r_m)w = \mathcal{A}_r(f)$, as needed.

□

Example 7.19. Consider the headless right-aligned formula tape with nonempty intermediary walls $f = 100(100)0000(0)$. Then the Algorithm 8 proceeds as follows:

$$\begin{aligned} t_0 &= 100\ 0000 \\ t_1 &= 100\ 100\ 0000\ 0 \\ t_2 &= 100\ 100\ 100\ 0000\ 0\ 0 \\ f_{\text{out}} &= \varnothing \end{aligned} \quad \rightarrow_{\text{Algorithm 8 goes to}} \quad \begin{aligned} t_0 &= 0000 \\ t_1 &= 100\ 0000\ 0 \\ t_2 &= 100\ 100\ 0000\ 0\ 0 \\ f_{\text{out}} &= 100 \end{aligned} \quad \rightarrow_{\text{Algorithm 8 goes to}} \quad \begin{aligned} t_0 &= 0000 \\ t_1 &= 0000\ 0 \\ t_2 &= 0000\ 0\ 0 \\ f_{\text{out}} &= 100(100) \end{aligned}$$

$$\begin{aligned} t_0 &= 0000 \\ t_1 &= 0000\ 0 \\ t_2 &= 0000\ 0\ 0 \\ f &= 100(100) \end{aligned} \quad \rightarrow_{\text{Algorithm 8 goes to}} \quad \begin{aligned} t_0 &= \varnothing \\ t_1 &= 0 \\ t_2 &= 0\ 0 \\ f_{\text{out}} &= 100(100)0000 \end{aligned} \quad \rightarrow_{\text{Algorithm 8 goes to}} \quad \begin{aligned} t_0 &= \varnothing \\ t_1 &= \varnothing \\ t_2 &= \varnothing \\ f_{\text{out}} &= 100(100)0000(0) \end{aligned}$$

Hence, the output is f, as claimed in Theorem 7.18.

Note that in some case, Algorithm 8 is able to fit formula tapes that have some empty intermediary walls, such as $f = 10110(110)(101)$. But in other cases, it cannot, such as with $f = 01(1)(0111)1$, which raises failure.

7.2.2 Bouncers decider

We finally piece all the elements of this chapter together and describe a decider for bouncers (excluding cyclers, decided in Section 2), Algorithm 9 which is proven correct in Theorem 7.22.

In order to drastically limit the amount of plausible subsequences to check, we limit our interest to *record-breaking* formula tapes, which are formula tapes where the head is pointing at 0^∞. In order to fit them, we can simply track record-breaking tapes that share the same head state and direction, i.e. tapes where the head is pointing at 0^∞. We first show that this is without loss of generality:

Lemma 7.20. If M is a bouncer that is not a cycler, solved by a formula tape f, then there is a record-breaking formula tape that also solves it.

Proof. We have $f \vdash_{\mathcal{A}} f_1 \vdash_{\mathcal{A}} f_2 \cdots \vdash_{\mathcal{A}} f_n$ with $n \geq 1$ and f_n is a special case of f. There is $1 \leq i \leq n$ such that f_i is record-breaking otherwise, the formula tapes do not grow and M is a cycler, which is excluded. Because f_n is a special case of f, it will eventually reach, under applications of $\vdash_{\mathcal{A}}$, a formula tape f' that is special case of f_i and we have the result. □

Lemma 7.21. If f is a formula tape with nonempty intermediary walls, then $\text{sync}(f)$ built in Theorem 7.12 has nonempty intermediary walls.

Proof. The construction of $\text{sync}(f)$ only involves (1) removing some repeaters (2) replacing some repeaters by some powers of themselves (3) increasing the size of some walls. Hence, $\text{sync}(f)$ has nonempty intermediary walls. □

Algorithm 9 proceeds by (1) tracking record-breaking tapes with same head (i.e. same state and pointing-direction) that grow linearly in quadratic time, (2) fitting formula tapes from triples from these plausible subsequences until a formula tape solves the bouncer is found or some limits are met.

Theorem 7.22 (Deciding bouncers)**.** Let M be a bouncer (Definition 7.8). Then, there exists a step limit $s \in \mathbb{N}$, a macro step limit $m \in \mathbb{N}$ and a formula tape testing limit l such that Algorithm 9, DECIDER-BOUNCERS(M,s,m,l) outputs **true**.

Proof. Because M is a bouncer, using Lemma 7.20 we know that there is a record-breaking formula tape $\tilde{f}$ that solves the bouncer. By Lemma 7.16 we know that we can transform $\tilde{f}$ into f with nonempty intermediary walls and that also solves the bouncer.

Algorithm 9 enumerates all record-breaking tapes triple that grow linearly in quadratic time. If s and l are large enough, by Theorem 7.12, we know that it will, eventually meet a triple of the form:

$$t_0 = C_{\text{sync}(f)}(0)$$
$$t_1 = C_{\text{sync}(f)}(1)$$
$$t_2 = C_{\text{sync}(f)}(2)$$

By Lemma 7.21, $f' = \text{sync}(f)$ is with nonempty intermediary walls and, at line 23, Algorithm 9 will feed $(\mathcal{S}(C_{f'}(0)), \mathcal{S}(C_{f'}(1)), \mathcal{S}(C_{f'}(2)))$ to Algorithm 8, which, by Theorem 7.18, will output $\mathcal{S}(\mathcal{A}_r(\text{sync}(f)))$ which can be reconstructed into $\mathcal{A}_r(\text{sync}(f))$ at line 27. By Theorem 7.12, we know that $\text{sync}(f)$ also solves the bouncer, and by Lemma 7.14, $\mathcal{A}_r(\text{sync}(f))$ also solves the bouncer. Theorem 7.9 will be verified on $\mathcal{A}_r(\text{sync}(f))$ using **reaches_special_case** at line 28, which, by hypothesis will output **true**, and we have the result. □

Remark 7.23. While Theorem 7.22 assures us that the decider will be able to detect a bouncer in bounded time, the formula tapes found in practice by the algorithm can be smaller than the $\text{sync}(f)$ construction we used to prove the upper bound.

Remark 7.24 (Implementation details of Algorithm 9)**.** In Algorithm 9, we assume that we are given (1) a **get_record breaking_tapes** routine which returns the record-breaking tapes for each tape head $h \in \Delta$ (i.e. tape head state and pointing-direction), in increasing length (2) a **binary_search_len** routine which finds by binary search a tape of a given length, or returns **None** if it does not exist and (3) a **is_quadratic** which tests that a sequence of integers is in quadratic progression (for instance by computing second differences and testing they are constant), (4) routines **headless** and **attach_head** to manipulate tapes/formula tapes with and without head, and (5) a **reaches_special_case** routine on formula tapes which returns **true** if successive simulation and alignment of the formula tape, as described in Theorem 7.9, reaches a special case in a given amount of macro steps, where a macro step consists in

Algorithm 9 DECIDER-BOUNCERS

```
1: procedure bool DECIDER-BOUNCERS(TM machine, uint step_limit, uint macro_step_limit, uint
   max_formula_tapes)
2:     Map[TMHead,Vec[Tape]] record_breaking_tapes = get_record_breaking_tapes(machine,
   step_limit)
3:     for TMHead head in record_breaking_tapes do
4:         uint num_tested_formula = 0
5:         for uint i, Tape tape4 in record_breaking_tapes[head].enumerate() do
6:             if i < 3 then continue
7:             bool break_outer = false
8:             for uint j, Tape tape3 in record_breaking_tapes[head][:i].enumerate() do
9:                 if j < 2 then continue
10:                uint len_diff = tape4.len() - tape3.len()
11:                uint tape2_len = tape3.len() - len_diff
12:                Tape or None tape_2 = record_breaking_tapes[head][:i].binary_search_len(tape2_len)
13:                if tape_2 is None then
14:                    continue
15:
16:                uint tape1_len = tape2.len() - len_diff
17:                Tape or None tape_1 = record_breaking_tapes[head][:i].binary_search_len(tape1_len)
18:                if tape_1 is None then
19:                    continue
20:
21:                if not is_quadratic([tape_1.step,tape_2.step,tape_3.step,tape_4.step]) then
22:                    continue
23:
24:                try FormulaTape f = FITFORMULATAPE(tape_1, tape_2, tape_3).headless()
25:                if Failure was raised then
26:                    continue
27:
28:                f.attach_head(tape_1.head)
29:                if f.reaches_special_case(macro_step_limit) then
30:                    return true
31:
32:                num_tested_formula += 1
33:
34:                if num_tested_formula == max_formula_tapes then
35:                    break_outer = true
36:                    break
37:
38:            if break_outer then
39:                break
40:    return false
```

performing alignment followed by either performing a usual step or a shift rule step. We can note that when detecting shift rules (see Section 7.1.2), it is important to implement cycler detection since it is possible for a machine to cycle indefinitely on a finite tape.

7.3 Implementations and results

Here are the implementations of the decider that were realised:

1. Tony Guilfoyle's C++ initial implementation (does not use the theory presented in this section): `https://github.com/TonyGuil/bbchallenge/tree/main/Bouncers`
2. Iijil's Go implementation (does not use the theory presented in this section): `https://github.com/Iijil1/Bouncers`

3. savask's Haskell implementation (basis of the theory presented in this section): `https://gist.github.com/savask/888aa5e058559c972413790c29d7ad72`

4. mei's optimised Rust implementation, reproducing savask's: `https://github.com/meithecatte/busycoq/`. This implementation outputs certificates that are verified using Coq, more details about this approach will be given in future versions of this text.

5. Tristan Stérin's (cosmo's) Rust implementation, reproducing savask's and mei's: `https://github.com/bbchallenge/bbchallenge-deciders/tree/main/decider-bouncers-reproduction`. This implementation follows this text to the letter, reproducing each concept and algorithm as presented here. It is less efficient than mei's.

Verifiers. Verifiers for Theorem 7.9, i.e. programs that verify bouncer certificates (Definition 7.10) have also been given as part of the above implementations. Mei's implementation provides a Coq implementation of the verifier. There is an ongoing effort for standardising the format of bouncers certificates (and certificates in general).

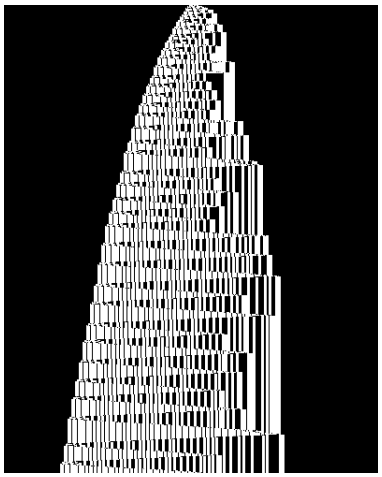


Figure 10: 50,000-step space-time diagram of `https://bbchallenge.org/5608043`. Out of the 29,799 decided bouncers, this bouncer takes the most steps (141,509) to be detected and fits the biggest formula tape, using Algorithm 9 presented in this section.

Remarkable bouncers. Out of the 29,799 bouncers that were decided using Algorithm 9, here are some remarkable facts:

1. Using Algorithm 9, only two bouncers with 3 repeaters were found (and that's the maximum): `https://bbchallenge.org/347505` and `https://bbchallenge.org/8131743`. Otherwise, 2132 bouncers have 2 repeaters and the rest have only 1.

2. The biggest fitted formula tapes by the algorithm have 328 symbols (summing walls and repeater symbols, not counting head and 0^∞), there are two of them, such as for machine `https://bbchallenge.org/5608043`, see Figure 10:

$$0^\infty \overset{A}{\triangleleft} 1000001110001110000001110000111011100111011100111000011$$
$$0000111011100111000011101110011101110011100000111000011101110$$
$$0$$
$$1110000111000011100001110000111011100111011100111011100111011$$
$$1001110111001110111001(111011100111011100111011100111011100)000$$
$$1110000111000000111001111111111111110000111(111111111111)00111$$
$$0000000111000000111111001111110^\infty$$

3. The above machine of Point 2 is also the machine that is detected after the most steps: 141,509. Over this dataset, it took 207 steps on average.

4. The most macro steps (i.e. number of usual or shift rule steps in formula tape simulation) needed to conclude using Theorem 7.9 was 41,628 for `https://bbchallenge.org/347505`. Otherwise, it took 66 macro steps on average.

References

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

[2] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *International Conference on Concurrency Theory*, pages 135–150. Springer, 1997.

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

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

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

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

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

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

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

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

[11] The bbchallenge Collaboration. Determination of the fifth Busy Beaver value. In submission, 2025.