Correctness of bbchallenge's deciders

bbchallenge's contributors

Abstract

The Busy Beaver Challenge (or bbchallenge) aims at collaboratively solving the following conjecture: "BB(5) = 47,176,870" [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 88,664,064 Turing machines with 5-state halt or not – starting from all-0 tape. In order to decide the behavior of these machines we write *deciders*. A decider is a program that takes as input a Turing machine and outputs `true` if it is able to tell whether the machine halts or not. Each decider is specialised in recognising a particular type of behavior that can be decided.

In this document we are concerned with proving the correctness of these deciders programs. More context and information about this methodology are available at `https://bbchallenge.org`.

Contents

1 Conventions

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

	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/1RB1LC1RC1RB1RD0LE1LA1LD---0LA&status=halt`

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 0, 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. Thus, the initial configuration of the machine can be written as $0^\infty\ [0]_A\ 0^\infty$.

Moreover, we will sometimes prefer to think of the tape head as being between symbols. Thus, we write $l <_q r$ to mean that the head is at the rightmost symbol of l, and $l >_q r$ to mean that the head is at the leftmost symbol of r. For example, $0^\infty\ [1]_A\ 0^\infty$ can be written as $0^\infty\ 1 <_A 0^\infty$ or $0^\infty >_A 1\ 0^\infty$.

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

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

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

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

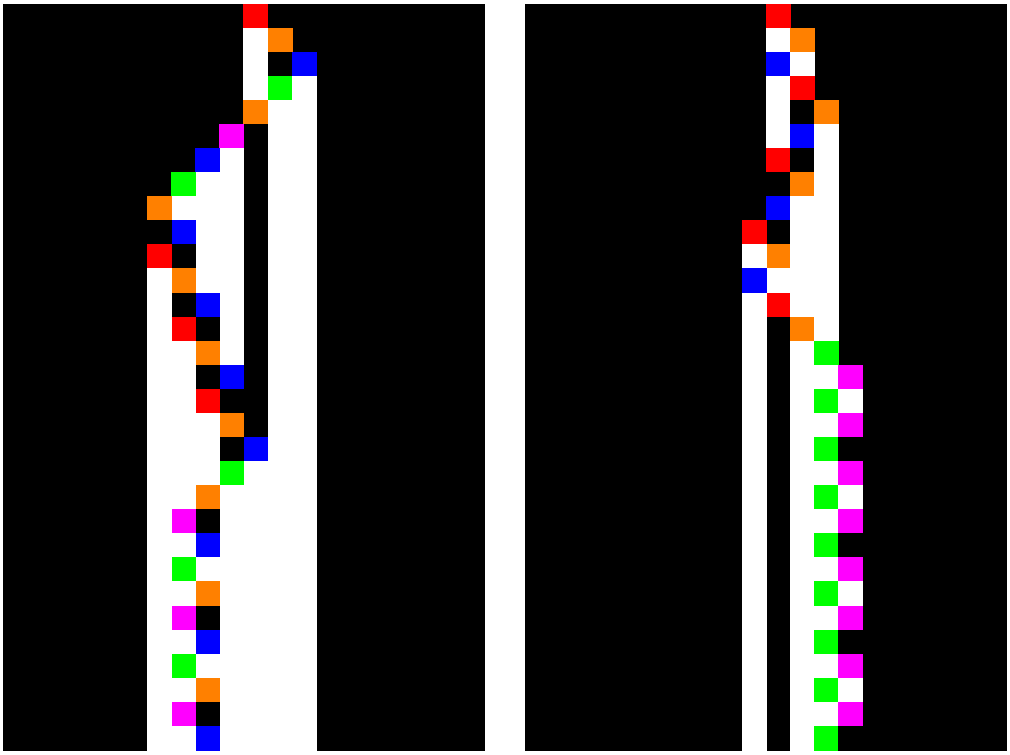


Figure 1: Space-time diagrams of the 30 first steps of bbchallenge's machines #279,081 (left) and #4,239,083 (right) which are both "Cyclers": they eventually repeat the same configuration for ever. Access the machines at `https://bbchallenge.org/279081` and `https://bbchallenge.org/4239083`.

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.

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 $c_0 \vdash^i c_i \vdash^j c_i$ 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 intermediate configurations below time 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 $c_0 \vdash^i c_i \vdash^j c_i$ with $i + j \leq t$, which is what we want. Index i corresponds to the first time that c_i is seen (l.13 in Algorithm 1) while index j corresponds to the second time that c_i is seen (l.11 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 $c_0 \vdash^i c_i \vdash^j c_i$ with $i + j \leq t$. It follows that for all $k \in \mathbb{N}$, $c_0 \vdash^{i+kj} c_i$. The machine never halts as it will visit c_i infinitely often. □

2.3 Results

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

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

Algorithm 1 DECIDER-CYLERS

```
1: struct Configuration {
2:     int state
3:     int headPosition
4:     int → int tape
5: }
6:
7: procedure bool DECIDER-CYLERS(TM machine,int timeLimit)
8:     Configuration currConfiguration = {.state = 0, .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
```

3 Translated cyclers

The goal of this decider is to recognise Turing machines that translate a bounded pattern for ever. 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.

The translated cycler of Figure 2 features a relatively simple repeating pattern and transient pattern (pattern occurring before the repeating patterns starts). These can get significantly more complex, bbchallenge's machine #59,090,563 is an example see Figure 3 and `https://bbchallenge.org/59090563`. The method for detecting the behavior is the same but more resources are needed.

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).

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

Algorithm 2 DECIDER-TRANSLATED-CYLERS

```
1: const int RIGHT, LEFT = 0, 1
2: struct ValueAndLastTimeVisited {
3:     int value
4:     int lastTimeVisited
5: }
6: struct Configuration {
7:     int state
8:     int headPosition
9:     int → ValueAndLastTimeVisited tape
10: }
11:
12: procedure bool DECIDER-TRANSLATED-CYLERS(TM machine,int timeLimit)
13:     Configuration currConfiguration = {.state = 0, .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 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
```

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

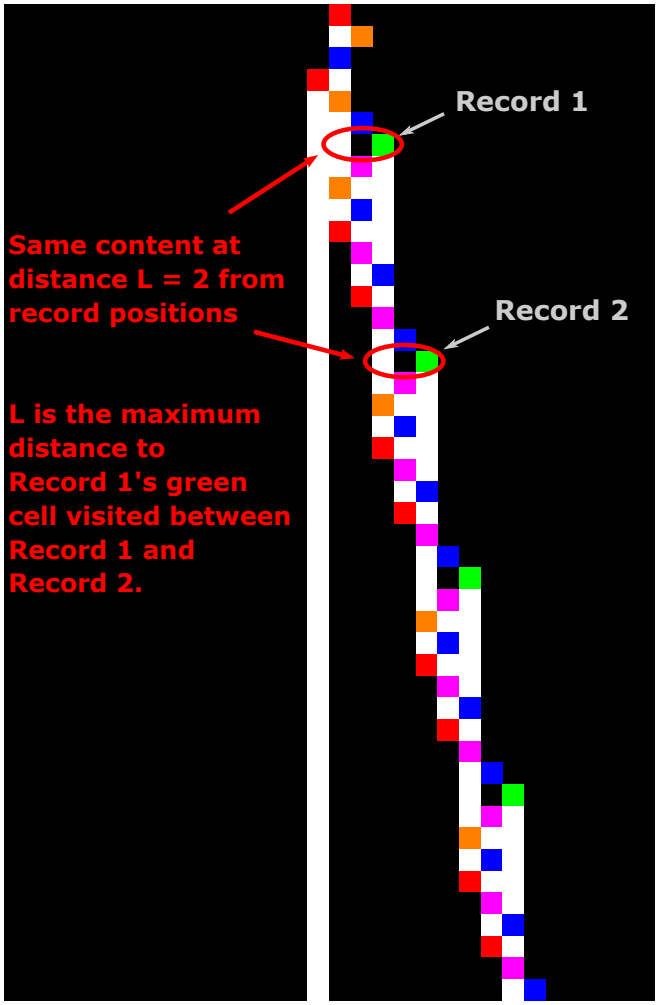


Figure 2: Example "Translated cycler": 45-step space-time diagram of bbchallenge's machine #44,394,115. See `https://bbchallenge.org/44394115`. The same bounded pattern is being translated to the right for ever. The text annotations illustrate the main idea for recognising "Translated Cyclers": find 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.

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 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 tape in r_1 at distance L of p_1 is the same than the content of the tape in r_2 at distance L of p_2 then $\mathcal{M}$ never halts. 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.

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

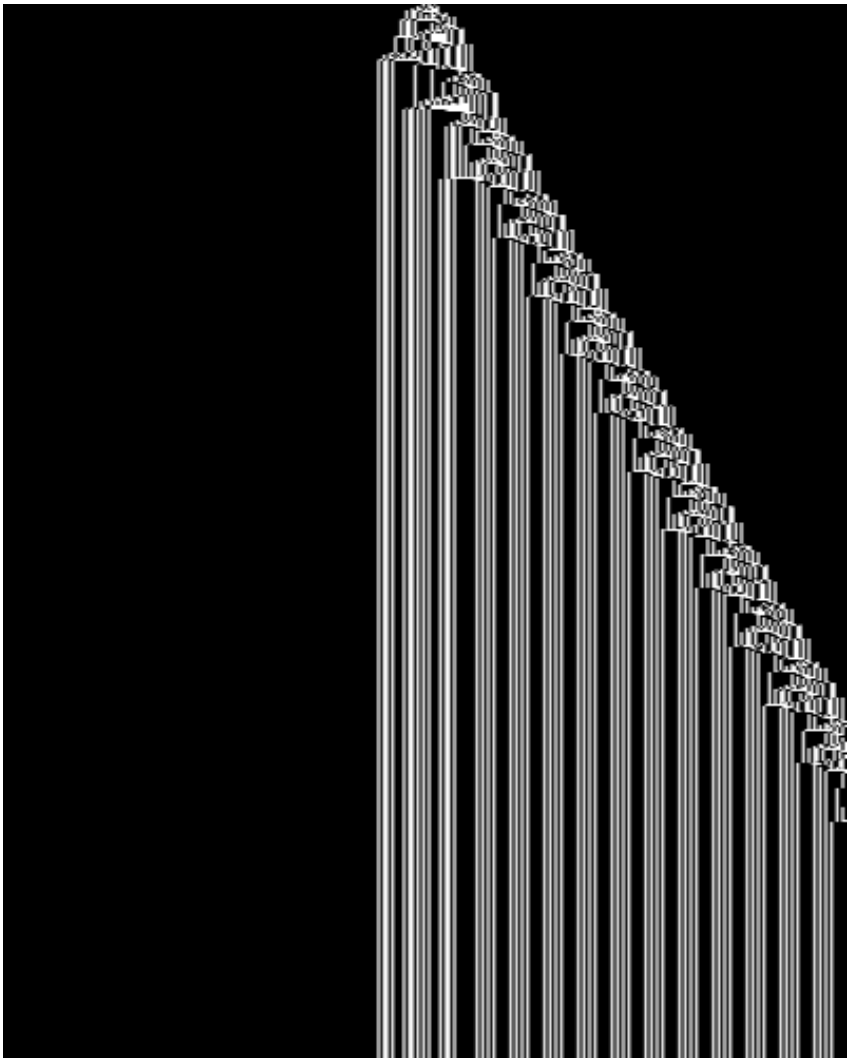


Figure 3: More complex "Translated cycler": 10,000-step space-time diagram (no state colours) of bbchallenge's machine #59,090,563. See `https://bbchallenge.org/59090563`.

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. Hence that after r_2, since it will read the same tape content the machine will reproduce the same behavior than it did after r_1 but translated at position p_2: there will 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 for ever 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.17) simulates the machine with the particularity that (a) it keeps track of the last time it visited each memory cell (l.19) and (b) it keeps track of all record-breaking configurations that are met (l.20) 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.12) loops over all older record-breaking configurations on the same side than the current one (l.13). The routine ignores older record-breaking configurations that were not in the same state than the current one (l.14). If the states are the same, it computes distance L (Definition 3.2) between the older and the current record-breaking configuration (l.16). 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.22). 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. □

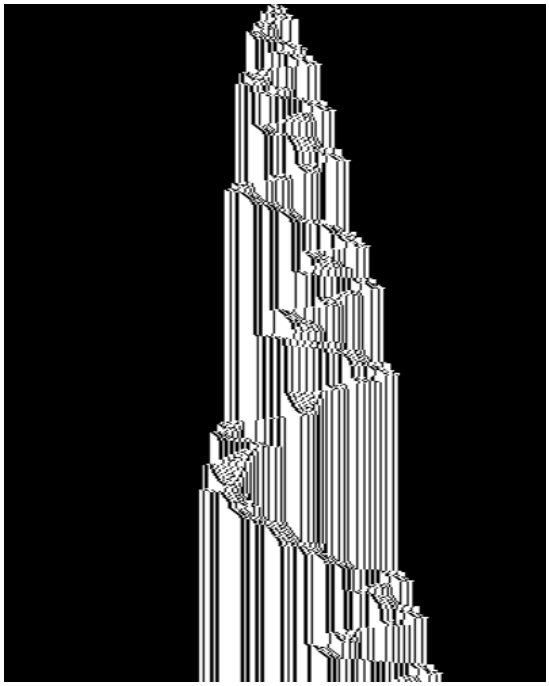
3.3 Results

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

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

4 Backward Reasoning

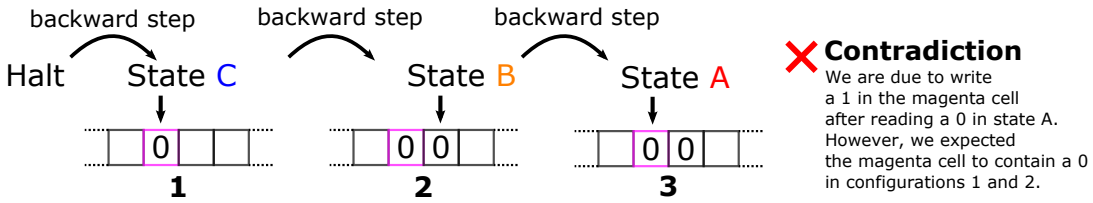
Backward reasoning, as described in [5], takes a different approach than what has been done with deciders in Sections 2 and 3. Indeed, instead of trying to recognise a particular kind of machine's behavior, the idea of backward reasoning is to show that, independently of the machine's behavior, the



(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 does 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 can do at least 20,000 without halting starting from all-0 tape.

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 com from 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 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 – e.g. 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

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.24). 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.25-28.

Then, the main loop runs until either (a) the stack is empty or (b) one leaf exceeded the maximum allowed depth, l.30 and l.32. 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 loop, we remove the current configuration from the stack and we try to apply all the transitions that leads to its state 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 the current transition's write instruction. This logic is checked 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.39).

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.

This pseudocode contains a slight optimisation with the use of set `configurationSeen` (l.29). This set racks configurations which would have already been seen in different branches of the tree in order not traverse them twice (l.32-33). While not needed in theory, this optimisation is useful in practice, especially at large depths (e.g. $D = 300$). □

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.

Algorithm 4 DECIDER-BACKWARD-REASONING

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

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 Results

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[2].

The decider decided 2,035,598 machines, out of 3,574,222 machines that were left after deciders for "Cyclers" and "Translated Cyclers" (Section 2 and Section 3.4). Maximum depth was set to 300. More information about these results are available at: `https://discuss.bbchallenge.org/t/decider-backward-reasoning/35`.

5 Halting Segment

Acknowledgement. Sincere thanks to bbchallenge's contributor Iijil who initially presented this method and the first implementation[3]. 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

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[4] `https://bbchallenge.org/1RB1RC_0LA0RA_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.
- Initial nodes (blue outline) correspond to all sgement 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.

[2]Thanks to collaborators `https://github.com/atticuscull` and `https://github.com/modderme123`.
[3]See: `https://discuss.bbchallenge.org/t/decider-halting-segment`.
[4]We chose a 3-state machine in order to have a graph of reasonable size.

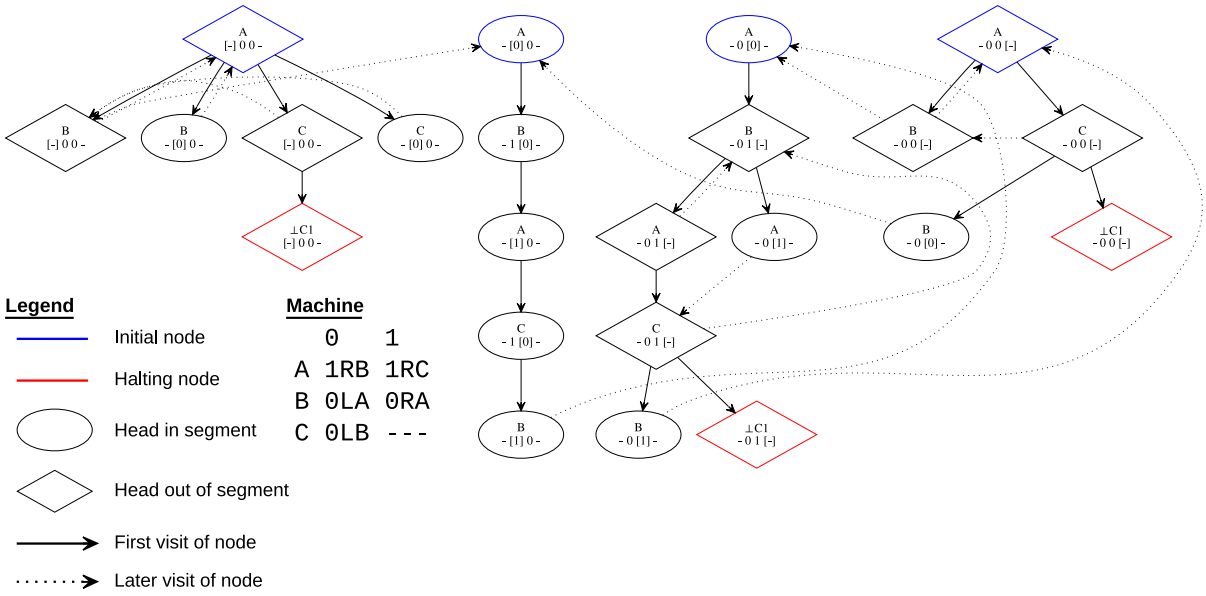


Figure 5: Halting Segment graph for the 3-state machine `https://bbchallenge.org/1RB1RC_0LA0RA_0LB---` and segment size 2, see Definition 5.3. Nodes of this graph correspond to *segment configurations* (Definition 5.1), i.e. configurations of the machine on a finite segment (here, of size 2). In a node, 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 readibility, 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.

- 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 size, 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 postions are not covered).

Interestingly, Halting Segment is the method that was used by Newcomb Greenleaf to prove[5] that Marxen & Buntrock's chaotic machine[6] [5] does not halt.

5.2 Formal proof

Definition 5.1 (Segment configurations)**.** Let $n \in \mathbb{N} = \{0, 1, 2 \dots\}$ a natural number called *segment size*. 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 $\perp$ and with an additional information (iv) of which halting transition of the machine has been used to halt.

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

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 $\perp$ `C1 - 0 0 [-]` (i) state $\perp$, (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 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.
- 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 $\perp$ `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 $\perp$ `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-enters 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 2) 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 $\perp$ `C1 [-] 0 0 -`, $\perp$ `C1 - 0 1 [-]` and $\perp$ `C1 - 0 0 [-]` which does not cover the entire segment (both internal segment postions 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 and results

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 than 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-reproduct` (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)

We will be only discussing the details and results of Iijil's implementation (1) as it was the first implementation to be proposed and that it also was reproduced independently with exactly matching results.

This implementation 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 than 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.

Nonetheless, results are impressive, for $n_{\texttt{max}} = 13$, the method decides 1,002,808 machines out of the 1,538,624 remaining after backward reasoning (see Section 4.3). Hence, after Halting Segment, we have 535,816 machines left to be decided[7].

[7] In fact 535,801 because 15 additional translated cyclers were decided, including some Skelet's machines.

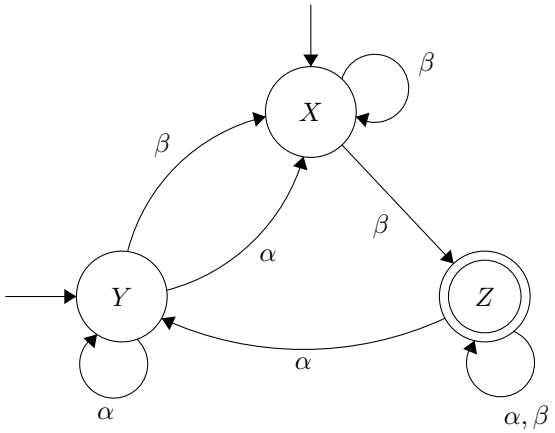


Figure 6: Example Nondeterministic Finite Automaton (NFA) with 3 states X,Y and Z, alphabet $\mathcal{A} = \{\alpha, \beta\}$, initial states X and Y, and accepting state Z. The linear-algebra representation of this NFA is given in Example 6.1. Example accepted words are: β, $\alpha\beta$, $\alpha\alpha\beta\beta$.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

6 Finite automata reduction

Acknowledgement. Sincere thanks to bbchallenge's contributor Justin Blanchard who initially presented this method and the first implementation[8]. Others have contributed to this method by producing alternative implementations (see Section 6.5) or discussing and writing the formal proof presented here: Tony Guilfoyle, Tristan Stérin (cosmo), Nathan Fenner, Mateusz Naściszewski (Mateon1), Konrad Deka, Iijil, Shawn Ligocki.

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 [1] 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[9]
2. C is *closed* under transitions: for any $c \in C$, the configuration one step later belongs to C[9]
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 [1] [4].

Here, we develop an original *co-CTL* technique[10], 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).

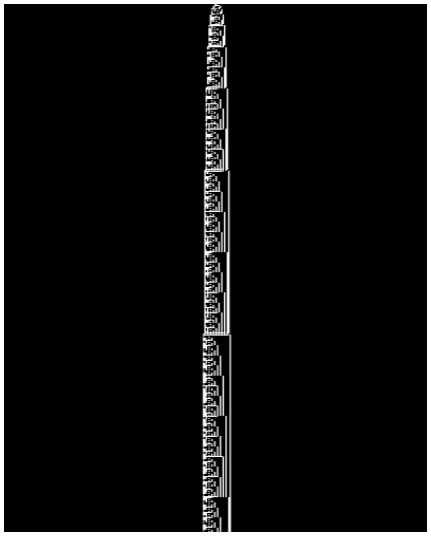
One important aspect of the technique is that, given a Turing machine and its constructed NFA—if found—it is a computationally simple task to verify that the NFA's language does indeed recognise all eventually-halting configurations (with finitely many 1s) of the machine.

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

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

[10] 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.

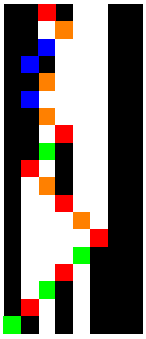
[11] `https://bbchallenge.org/1RB0LD_1LC1RA_0RB0LC_---1LA`, the machine exhibits a non-trivial counting behavior.



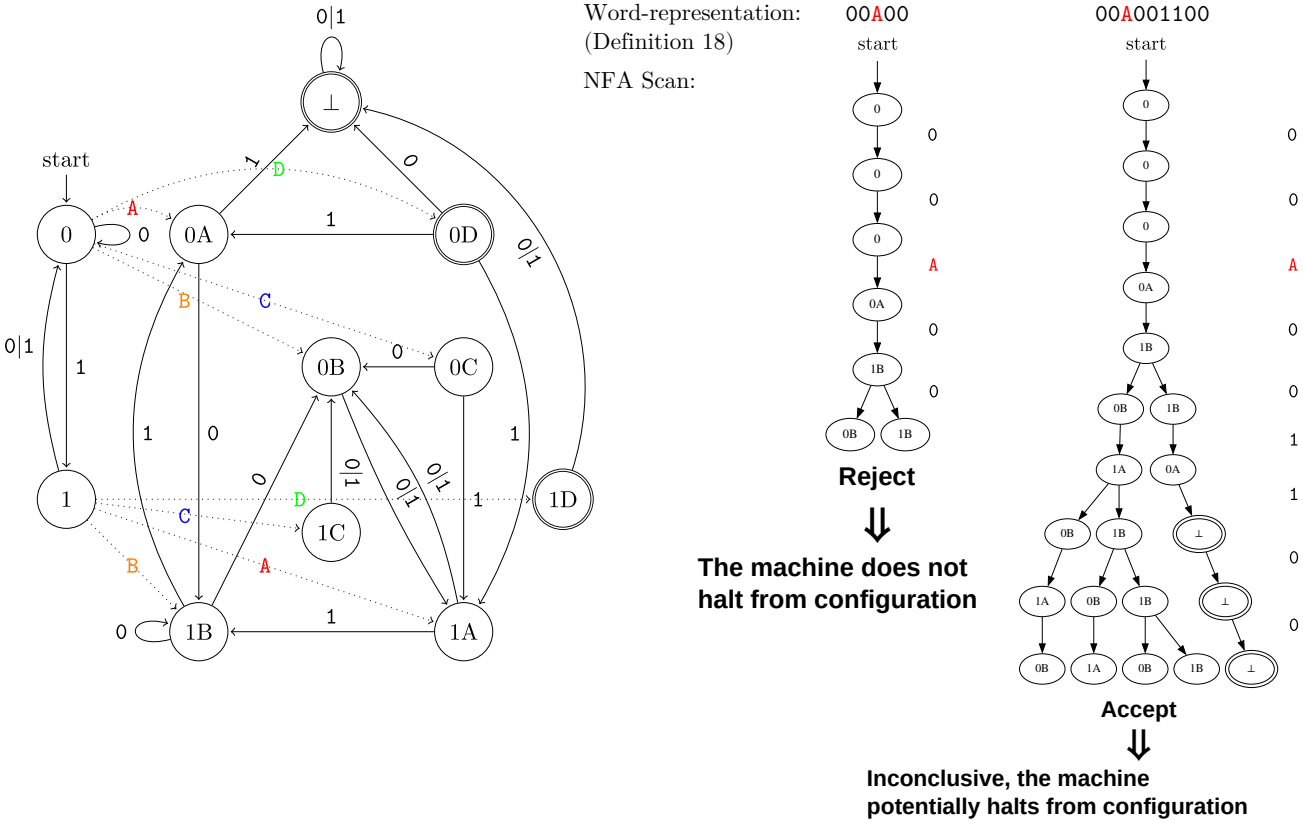
(a) 10,000-step space-time diagram of the 4-state Turing machine given in (b) from the all-0 initial configuration. The machine does not halt from the all-0 configuration.

	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[11]: (a) Space-time diagram showing the first few descendants of the all-0 configuration for the machine. The machine actually runs for ever from the all-0 configuration, adopting a "counting" behavior. (b) The TM halts in 18 steps from a different configuration; these 18 rows depict *eventually-halting* configurations. (c) Transition table for the TM. (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 (b), encoded as word `000A01100` (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 `000A01100`, 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 Automta (**NFA**) can be described using linear algebra. Let $\mathbf{2}$ denote the Boolean semiring[12] $\{0,1\}$ with operations $+$ and $\cdot$ respectively implemented by OR and AND [3]. 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 head's state (a letter between A and E in the case of 5-state TMs) just before the tape symbol its currently reading.

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 ask 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 $T_0 \in \mathcal{M}_{n,n}$:

$$q_0 T_0 = q_0 \tag{1}$$
$$T_0 a^T = a^T \tag{2}$$

Note that Condition 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.

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

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

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

With c, c_1, c_2 configurations of the TM (with finite support) and $\hat{c}, \hat{c}_1, \hat{c}_2$ any of their finite word-representations, see Definition 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:

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

Which algebraically becomes:

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

These conditions are unwieldy. Let's seek stronger (thus still sufficient) conditions which are simpler:

- For machine transitions going left/right, simply require $T_t T_b T_w \preceq T_b T_f T_r$ and $T_w T_t \preceq T_f T_r$, respectively with $\preceq$ the following relation on same-size matrices: $M \preceq M'$ if $M_{ij} \leq M'_{ij}$ element-wise, that is, if the second matrix has at least the same 1-entries as the first matrix.
- To simplify the condition for halting machine transitions: define an *accepted steady state-set* s to be a row vector such that $sa^T = 1$, $sT_0 \succeq s$, and $sT_1 \succeq s$. Given such s, we have that: $\forall q \in \mathcal{M}_{1,n} \ q \succeq s \implies \forall z \in \{0,1\}^* : qT_z a^T = 1$. Assuming that such s exists we can simply require: $\forall u \in \{0,1\}^* : q_0 T_u T_f T_r \succeq s$ which is stronger than $\forall u, z \in \{0,1\}^* : \ q_0 T_u T_f T_r T_z a^T = 1$ with $(f, r) \to \bot$ a halting transition.

Combining the above, we get our main result:

Theorem 6.4. Machine $\mathcal{M}$ doesn't halt from the initial all-0 configuration if there are an NFA $(q_0, \{T_\gamma\}, a)$ and row vector s satisfying the below:

$$q_0 T_0 = q_0 \qquad \text{(leading zeros ignored)} \qquad (1)$$
$$T_0 a^T = a^T \qquad \text{(trailing zeros ignored)} \qquad (2)$$
$$sa^T = 1 \qquad (s \text{ is accepted}) \qquad (3)$$
$$sT_0, sT_1 \succeq s \qquad (s \text{ is a steady state}) \qquad (4)$$
$$\forall u \in \{0,1\}^* : q_0 T_u T_f T_r \succeq s \qquad \text{if } (f, r) \to \bot \text{ is a halting transition of } \mathcal{M} \qquad (5)$$
$$T_b T_f T_r \succeq T_t T_b T_w \qquad \text{if } (f, r) \to (t, w, \text{left}) \text{ is a transition of } \mathcal{M} \qquad (6)$$
$$T_f T_r \succeq T_w T_t \qquad \text{if } (f, r) \to (t, w, \text{right}) \text{ is a transition of } \mathcal{M} \qquad (7)$$
$$q_0 T_A a^T = 0 \qquad \text{(initial configuration rejected)} \qquad (8)$$

Proof. Conditions (1)–(7) ensure that the NFA's language includes at least all eventually halting configurations of $\mathcal{M}$. Condition (8) ensures that the initial all-0 configuration of the machine is rejected, hence not eventually halting. Hence, if conditions (1)–(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 (1)–(7) hold and thus that the TM does not halt, is computationally simple[13]. Verifiers have been implemented for Theorem 6.4, see Section 6.5.

[13] Note that although equation (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 (5) replacing $q_0 T_u$ per such state.

Now, we want to design an efficient search algorithm that will, for a given TM, try to find an NFA satisfying Theorem 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 (1)–(7) is automatically satisfied by these NFAs.

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 of 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 (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, …, $e_{5l} \in \mathcal{M}_{1,d}$ 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 $e_{5l+1} = e_\perp$ is 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 (1)–(7) for R_0, R_1, and a. If Condition (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 (4)–(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[14].

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

$$R_r \succeq (e_\perp)^T e_\perp \qquad \text{for } r \in \{0,1\} \tag{4'}$$

$$\forall i \in [l] : R_r \succeq \mathrm{row}_i(M_f)^T e_\perp \qquad \text{if } (f,r) \to \perp \text{ is a halting transition of } \mathcal{M} \tag{5'}$$

$$\forall b \in \{0,1\}, \forall i \in [l] : R_r \succeq \mathrm{row}_{\delta(i,b)}(M_f)^T \mathrm{row}_i(M_t) R_b R_w \qquad \text{if } (f,r) \to (t,w,\text{left}) \text{ is a transition of } \mathcal{M} \tag{6'}$$

$$\forall i \in [l] : R_r \succeq \mathrm{row}_i(M_f)^T \mathrm{row}_{\delta(i,w)}(M_t) \qquad \text{if } (f,r) \to (t,w,\text{right}) \text{ is a transition of } \mathcal{M} \tag{7'}$$

Lemma 6.7. There's a unique minimal solution (w.r.t $\preceq$) to the system of inequalities (4')–(7') and an effective way to compute it: initialize R_0, R_1 to zero, then set entries to 1 as (4'), (5') and (7') demand then iterate (6') until R_0 and R_1 stop changing.

Proof. First notice that (4'), (5') and (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') 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 (2), (3) of Theorem 6.4 translate to:

$$R_0 a^T = a^T \tag{2'}$$

$$a \succeq e_\perp \tag{3'}$$

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 (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 (2') and (3').

The last condition from Theorem 6.4 that we need to satisfy is (8) (rejection of the initial configuration), which translates to:

$$\mathrm{row}_0(M_A) a^T = 0 \tag{8'}$$

By minimality, a solution of (2') and (3') will satisfy (8') if and only if the minimal solution exhibited above does. Hence, we check (8') for the minimal a that we constructed and there are two cases:

- If a satisfies (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 (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 (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 transitions functions.

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 (4') and (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 (7'), with (i, w) = divmod(k-1, 2)
12:        repeat
13:            Increase R[k] using (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)
```

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. [6]

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

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

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

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

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

Lemma 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 (1) or $\delta(0,0) = 0$);
2. its transition table includes each of $0, \ldots, n-1$, whose first appearances occur in order, and with each $0 < q < n$ appearing before the $2q$ position in the transition table;
3. the sequence $\{m_k := \max\{\delta(q, b) : 2q + b \le k\}\}_{k=0}^{2n-1}$ of cumulative maxima runs from 0 to $n-1$ in steps of 0 or 1, with $m_{2q-1} \ge q$ for $0 < q < n$.

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

2 $\iff$ 3**:** If the first appearances of $0, \ldots, n-1$ appear in order, any value at its first index is the largest so far, so m_k takes the same values. The sequence m_k is obviously nondecreasing, so to be gap-free

it can only grow in steps of 0 or 1. Conversely, if m_k runs from 0 to $n-1$ in steps of 0 or 1, each value $q \in [n]$ must appear in the table at the first index k for which $m_k = q$, and all preceding values in the transition table must be strictly less.

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

□

Corollary 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 $m_{2q-1} \geq q$ (for all $2q - 1 \leq \ell$).

Proof. If $\ell = 2n - 1$, $\{m_k\}$ grows to exactly $n - 1$ (since $m_{2(n-1)-1} \geq n - 1$), and lemma 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 Implementations and results

Here are the implementations of the decider that were realised:

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

Verifiers for Theorem 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`.

Results. In order to achieve reasonable compute time, the DFA search space (see Section 6.3) was searched up to 6 DFA states. The method decides 503,169 machines out of the 535,801 remaining machines (94%) after halting segment, in a bit less than 30 minutes using Justin Blanchard's rust implementation on a 4-core i7 laptop. Hence, after FAR, we have 32,632 machines left to be decided.

7 Bouncers

Intuitively, a *bouncer* is a Turing machine that builds a tape out of ever-expanding repeating fragments. The decider analyses the machine to identify the growing *repeaters* and the *walls* that separate them, and then executes the machine symbolically to prove that this growth is endless.

Notation. For a tape segment $s \in \{0,1\}^*$, we will write (s) to mean s repeated any number of times, i.e. s^k for an unspecified k.

During the

Remark 7.1. The symbolic tapes we're considering can also be understood as defining a regular language of tapes matching a particular step of the cycle being simulated. In this interpretation, we are executing the Turing machine on all elements of the language at the same time.

Specifically, we will be considering a tape language of the form

$$\{0^\infty\ [0]_q\ w_0\ (r_1)^{k_1}\ w_1\ (r_2)^{k_2}\ w_2\ \cdots\ (r_n)^{k_n}\ w_n \mid k_1, k_2, \ldots, k_n \in \mathbb{N}\}. \tag{9}$$

for a fixed list of *walls* $w_0,\ \ldots,\ w_n \in \{0,1\}^*$, a list of *repeaters* $r_1,\ \ldots,\ r_n$, and a fixed state q.

Fundamentally, the technique is a variant of Closed Tape Languages, but To prove that a Turing machine is a bouncer, we exhibit a set of configurations More formally, we say that a Turing machine is a bouncer if We shall refine this definition later, such that it clearly explains what machines the decider should hope to decide.

Example 7.2. 0(110)1 can refer to 01, 01101, 01101101, and so on.

Definition 7.3. A bouncer's *reset configuration* is given by and refers to the configuration

References

[1] Shawn ligocki blog: Ctl filter. `https://www.sligocki.com/2022/06/10/ctl.html`. Accessed: 2023-03-20.

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

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

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

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

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