Correctness of bbchallenge's deciders

Tristan Stérin

Abstract

The Busy Beaver Challenge (or bbchallenge) aims at collaboratively solving the following conjecture: "BB(5) = 47,176,870" [Aaronson, 2020]. This goal 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

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

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`

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 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 $c_1 \vdash \bot$ 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.

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 Decider for "Cyclers"

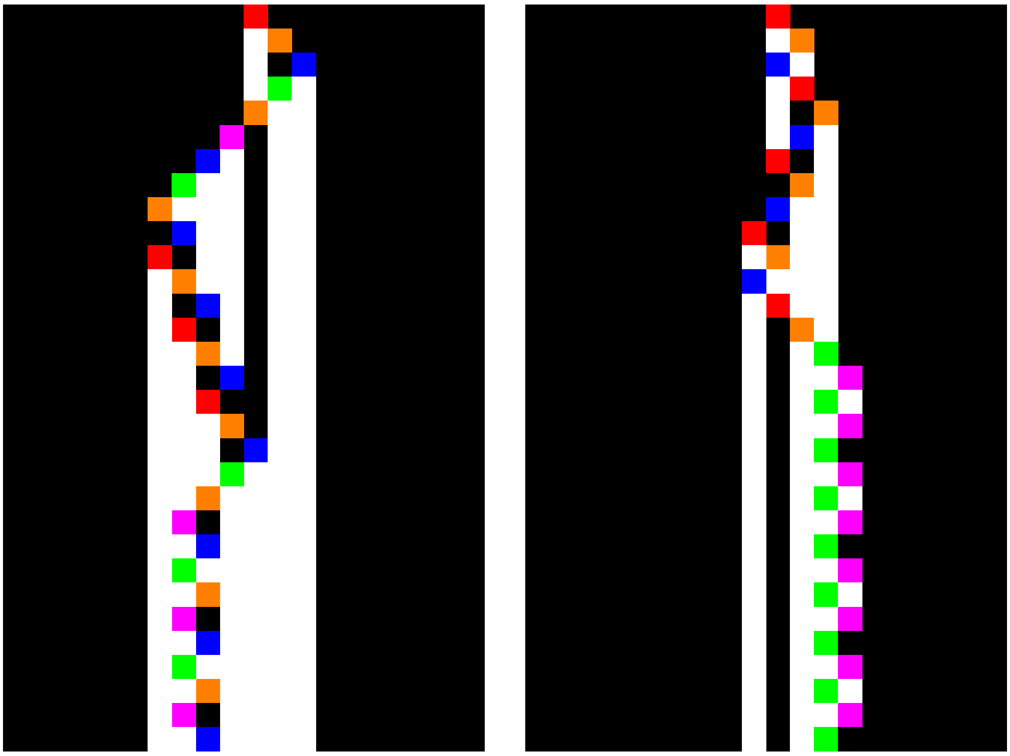


Figure 1: Space-time diagrams of the 30 first steps of bbchallenge's machines #279,081 (left) and #4,239,083 (right) which are both "Cyclers": they eventually repeat the same configuration for ever. Access the machines at `https://bbchallenge/279081` and `https://bbchallenge/4239083`.

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 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/279081` and `https://bbchallenge/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.

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.

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
```

2.2 Correctness

Theorem 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 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, 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`). More information about these results are available at: `https://discuss.bbchallenge.org/t/decider-cyclers/33`.

3 Decider for "Translated cyclers"

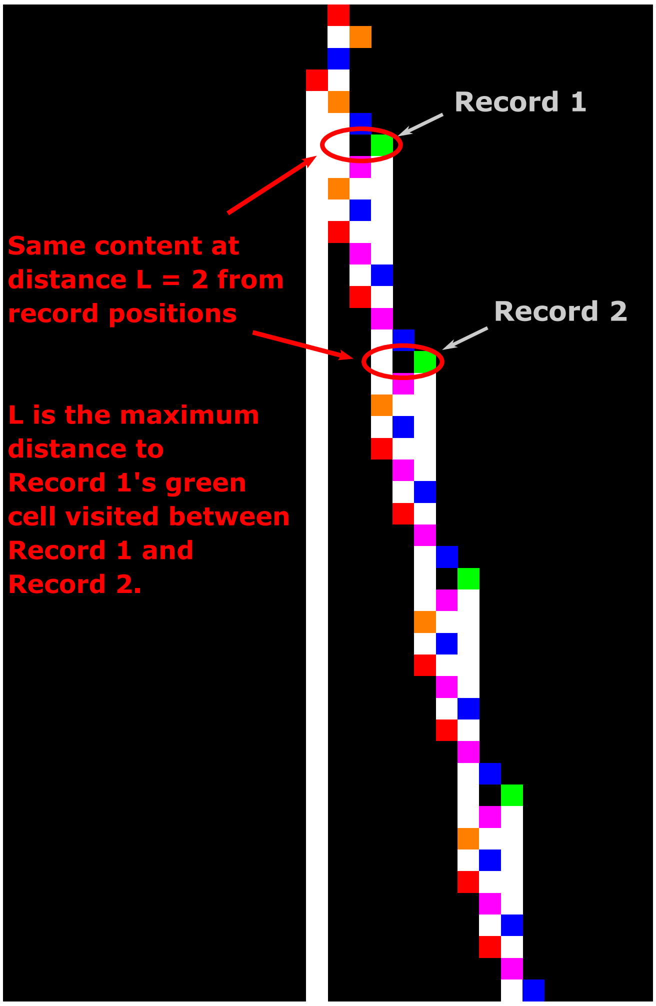


Figure 2: Example "Translated cycler": 45-step space-time diagram of bbchallenge's machine #44,394,115. See `https://bbchallenge.org/44394115`. The same bounded pattern is being translated to the right for ever. The text annotations illustrate the main idea for recognising "Translated Cyclers": find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

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.

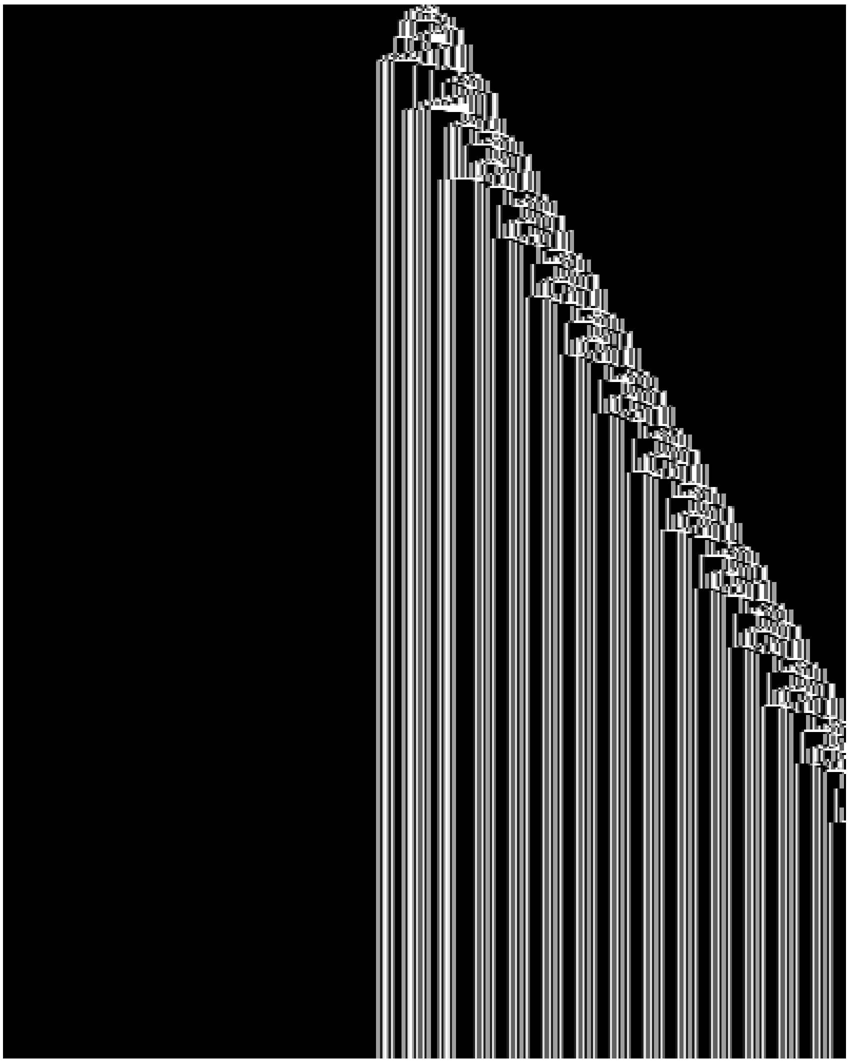


Figure 3: More complex “Translated cycler”: 10,000-step space-time diagram (no state colours) of bbchallenge’s machine #59,090,563. See `https://bbchallenge.org/59090563`.

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 5) it is useful to add to memory cells the information of the last time step at which it was visited.

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

Algorithm 2 DECIDER-TRANSLATED-CYLERS

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

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

3.2 Correctness

Definition 4 (record-breaking configurations)**.** Let $\mathcal{M}$ be a Turing machine and c_0 its busy beaver initial configuration (i.e. state is 0, head position is 0 and tape is all-0). Let c be a configuration reachable from c_0, i.e. $c_0 \vdash^* c$. Then c is said to be *record-breaking* if the current head position had never been visited before. Records can be broken to the *right* (positive head position) or to the left (negative head position).

Definition 5 (Distance L between record-breaking configurations)**.** Let $\mathcal{M}$ be a Turing machine and r_1, r_2 be two record-breaking configurations on the same side of the tape at respective times t_1 and t_2 with $t_1 < t_2$. Let p_1 and p_2 be the tape positions of these records. Then, distance L between r_1 and r_2 is

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

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

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

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

Lemma 6. Let $\mathcal{M}$ be a Turing machine. Let r_1 and r_2 be two configurations that broke a record in the same state and on the same side of the tape at respective times t_1 and t_2 with $t_1 < t_2$. Let p_1 and p_2 be the tape positions of these records. Let L be the distance between r_1 and r_2 (Definition 5). 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 5, we know that distance L is the maximum distance that $\mathcal{M}$ can travel to the left of p_1 between times t_1 and t_2.

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. 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 5). 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 7. Let $\mathcal{M}$ be a Turing machine and t a time limit. The conditions of Lemma 6 are met before time t if and only if DECIDER-TRANSLATED-CYCLERS($\mathcal{M}$,t) outputs `true` (Algorithm 2).

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

The main loop of DECIDER-TRANSLATED-CYCLERS (Algorithm 2 l.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 6. 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 5) 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 6, which is what we wanted.

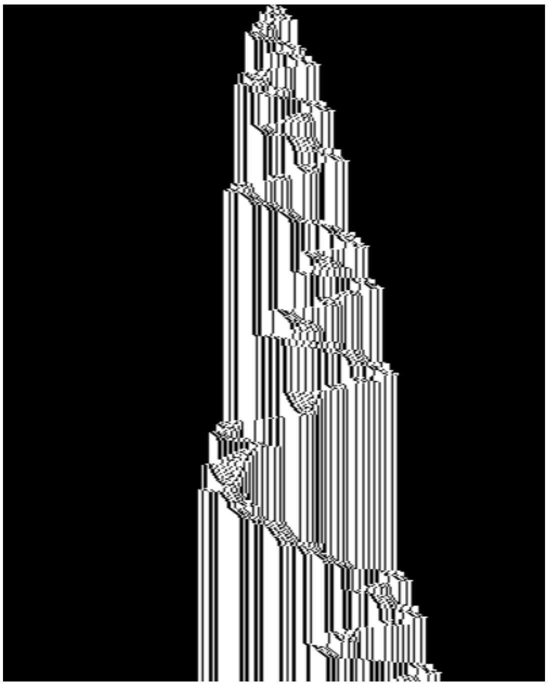
□

Corollary 8. Let $\mathcal{M}$ be a Turing machine and $t \in \mathbb{N}$ a time limit. If DECIDER-TRANSLATED-CYCLERS($\mathcal{M}$,t) returns `true` then the behavior of $\mathcal{M}$ from all-0 tape has been decided: $\mathcal{M}$ does not halt.

Proof. Immediate by combining Lemma 6 and Theorem 7. □

3.3 Results

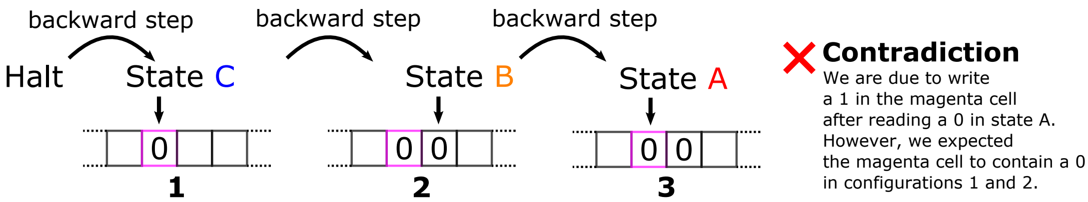
The decider was coded in `golang` and is accessible at this link: `https://github.com/bbchallenge/bbchallenge-deciders/tree/main/decider-translated-cyclers`.



(a) 10,000-step space-time diagram of bbchalenge'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.

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`). More information about these results are available at: `https://discuss.bbchallenge.org/t/decider-translated-cyclers/34`.

4 Decider for backward reasoning

Backward reasoning, as described in [2], 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 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

Algorithm 4 DECIDER-BACKWARD-REASONING

```
const int RIGHT, LEFT = 0, 1
struct Transition {
    int state, read, write, move
}
struct Configuration {
    int state
    int headPosition
    int → int tape
    int depth
}

procedure Configuration APPLY-TRANSITION-BACKWARDS(Configuration conf,Transition t)
    int reversedHeadMoveOffset = (t.move == RIGHT) ? -1 : 1
    int previousPosition = conf.headPosition+reversedHeadMoveOffset
    // Backward contradiction spotted
    if previousPosition in conf.tape and conf.tape[previousPosition] != t.write then
        return nil
    Configuration previousConf = {.state = t.state, .depth = conf.depth + 1, .tape = conf.tape}
    previousConf.headPosition = previousPosition
    previousConf.tape[previousPosition] = t.read
    return previousConf

procedure bool DECIDER-BACKWARD-REASONING(TM machine,int maxDepth)
    Stack<Configuration> configurationStack
    for int (state,read) in GET-UNDEFINED-TRANSITIONS(machine) do
        Configuration haltingConfiguration = {.state = state,.depth=0,.headPosition = 0}
        haltingConfiguration.tape = {0: read}
        configurationStack.push(haltingConfiguration)
    Set<Configuration> configurationsSeen = {}
    while !configurationStack.empty() and configurationStack.top().depth < maxDepth do
        Configuration currConf = configurationStack.pop()
        if currConf in configurationsSeen then continue
        configurationsSeen.insert(currConf)
        List<Configuration> confList = []
        for Transition transition in GET-TRANSITIONS-REACHING-STATE(machine,currConf.state) do
            Configuration previousConf = APPLY-TRANSITION-BACKWARDS(currConf, transition)
            // If no contradiction
            if previousConf != nil then
                configurationStack.push(previousConf)
    // Returns true iff all leaves at depth ≤ maxDepth reached a contradiction
    return configurationStack.empty()
```

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 9. 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) maximum depth has been reached, l.30. 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 10. 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 9 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. □

References

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

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