Brainfuck is Turing-complete

ziem24

September-October 2025

Abstract

Turing-completeness of a system is the ability to perform any calculation that can be computed by a Universal Turing Machine (UTM). In this paper, I propose a model that can simulate any Turing machine program using Brainfuck, with the main goal of creating a TM-to-Brainfuck compiler.

1 Introduction to Brainfuck

Brainfuck is one of the simplest esoteric programming languages. It consists of a memory tape with unsigned 8-bit values initialized with 0's, and a single pointer which starts at the position 0. If the pointer goes out of bounds, the program ends with an error. It has the following instruction set:

$+ \longrightarrow$ increment $[ptr]$ [1]
$- \longrightarrow$ decrement $[ptr]$
$> \longrightarrow$ increment ptr
$< \longrightarrow$ decrement ptr
$[\longrightarrow$ if $[ptr] = 0$, go to the matching $]$
$] \longrightarrow$ if $[ptr] \neq 0$, go to the matching $[$
$, \longrightarrow$ input a character at ptr
$. \longrightarrow$ output $[ptr]$

The last two instructions handle I/O, and thus will not be needed to show Brainfuck's Turing-completeness.

[1] In context of memory access, $[X]$ means "value under the address/pointer X"

2 Turing machines

Let $\mathcal{M} = \langle Q, q_0, q_h, \Gamma, \varepsilon, \delta \rangle$ define a single-tape Turing machine with states Q - where $q_0 \in Q$ is the initial state, and $q_h \in Q$ is the halting state - an alphabet $\Gamma : |\Gamma| = 2$, a blank symbol $\varepsilon \in \Gamma$, and a state transition function:
$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$. $\mathcal{M}$ has the same computational power as a UTM, meaning that a system is Turing-complete if it can simulate $\mathcal{M}$ (given sufficient memory space).

For practical purposes, let $\Gamma = \{0, 1\}$. We can encode every transition in a matrix $[T]_{|Q| \times 6}$, where for each row $T_q = [q_{0q}, s_{0q}, m_{0q}, q_{1q}, s_{1q}, m_{1q}]$, the state transition function δ can be expressed as

$$\delta(q, s) = \begin{cases} (q_{0q}, s_{0q}, m_{0q}) & \text{if } s = 0 \\ (q_{1q}, s_{1q}, m_{1q}) & \text{if } s = 1 \end{cases}$$

Below is an example of how that simulator may be implemented in C:

```c
#define LENGTH 50
#define H -1

int main() {
  int ptr = 24, tape[LENGTH], current_state = 0;
  for (int i = 0; i < LENGTH; i++) { tape[i] = 0; }
  // state_0, set_0, move_0, state_1, set_1, move_1
  int trans_0[6] = {1, 1,  1, 1, 1, -1};
  int trans_1[6] = {0, 1, -1, 2, 0, -1};
  int trans_2[6] = {H, 1,  1, 3, 1, -1};
  int trans_3[6] = {3, 1,  1, 0, 0,  1};
  int* transitions[] = {trans_0, trans_1, trans_2, trans_3};

  while (current_state != H) {
    int* current_t = transitions[current_state];
    if (tape[ptr] == 0) {
      current_state = current_t[0];
      tape[ptr] = current_t[1];
      ptr += current_t[2];
    }
    else {
      current_state = current_t[3];
      tape[ptr] = current_t[4];
      ptr += current_t[5];
    }
  }
}
```

3 Simulating a Turing machine in Brainfuck

To simulate any Turing machine in Brainfuck, we are going to need space for the tape, and a place to store every transition tuple T_i, and a structure (later referred to as C [2]) that holds the new state $\chi \in Q$, symbol $\sigma \in \{0,1\}$, and movement direction $\mu \in \{0,1\}$:

$$\mathbb{S} = [T_1, T_2, T_3, \ldots, T_n][C][TAPE] : n = |Q|, \text{ where}$$
$$TAPE \in \{0,1\}^M : M \in \mathbb{N}, \text{ and}$$
$$C = [\chi][\sigma][\mu] \in \{0,1\}^{|\chi|+2}$$

3.1 Changing the states

The first thing that the program will handle is changing the state. With the arbitrary amount of states, more than one cell may be needed to encode them. Let each state be a binary sequence of fixed length $|q| = k : k \in \mathbb{N}$. Note that k can be computed at compile time. Let a transition header be the structure that holds information about the new state, symbol, and movement direction. For now, this is how each header might look like:

$$T_q = [I][q_{01}, q_{02}, \ldots, q_{0k}][s_0][m_0][q_{11}, q_{12}, \ldots, q_{1k}][s_1][m_1] \in \{0,1\}^{|T_q|}, \text{ where}$$

- I - state index: if $I_{T_{q+1}} = 0$ and $I_{T_q} = 1$ then T_q is the targeted header;
- q_x - new state if value under the Turing machine head H is 1;
- s_x - new symbol if $[H] = x$;
- m_x - movement direction if $[H] = x$.

The length of this structure is equal to $\tau := |T_q| = 2k + 5$. Because C is placed to the right of the headers in $\mathbb{S}$, it would be easier if the headers were sorted in descending order, starting from T_n and ending at T_1. If we interpret χ as a binary number (big-endian), moving the pointer to the state T_χ requires us to change some number t of header indexes to 1, where:

$$t = \sum_{i=1}^{k} 2^{k-i} \chi_i$$

Going back from T_χ is fairly simple and can be implemented as $[\langle > \rangle^\tau]$, which moves the pointer from I_{T_χ} to the right, until it encounters 0. We, however, need to guarantee that the value under $(I_{T_1} + \tau)$ is 0. Let this cell be called Ω. This is how $\mathbb{S}$ looks like after applying all of the modifications:

$$\mathbb{S} = [T_n, \ldots, T_1][\Omega][C][TAPE]$$

[2] It is possible to simulate them without it, but the result will take up more memory, while being slower and more complex.

Let $S(i)$ be the procedure that starts at χ_i and increments the values of the first 2^{k-i} zeroed state indexes. Let $(\diamond|x) = \langle\diamond\rangle^x = \underbrace{\diamond\diamond\cdots\diamond}_{x \text{ times}}$, where $\diamond$ is a sequence of Brainfuck instructions. This is how $S(i)$ will be implemented:

$$S(i) = [\langle<\rangle^{i+\tau}\,[\langle<\rangle^{\tau}] + \langle\langle<\rangle^{\tau} +\rangle^{2^{k-i}-1}\,[\langle>\rangle^{\tau}]\,\langle>\rangle^{i} -]$$

The pointer starts and ends at χ_i. While $\chi_i \neq 0$ (practically, equal to 1), the pointer goes to I_{T_n}, after which it moves to the leftmost $I_T = 0$. Then it increments its value and the values of the next $2^{k-i} - 1$ indexes to the left, effectively increasing the state number by 2^{k-i}. After this, it returns to χ_i and decrements its value. Starting from Ω, this is the full implementation of the state change algorithm:

```
procedure SET_Q
    >>
    for i in [1, ..., k] do { S(i)> }   // ends at sigma
```

3.2 Replacing tape symbols

I will approach the problem of head positioning similarly to state indexing in 3.1. This is how the $TAPE$ could look like:

$$TAPE = [\Theta][s_0, J_0][s_1, J_1]\ldots[s_m, J_m] : m \in \mathbb{N}, \text{ where}$$

- $\Theta = 0$: serves the same purpose as Ω in $\mathbb{S}$
- J_i: tape index - if $J_i = 0$ and $J_{i-1} = 1$, the head is placed at $TAPE[i]$;
- s_i: tape symbol.

The procedure below starts at σ, ends at μ, and handles the symbol change:

```
procedure SET_S
    >>>>[>>]<[-]<[<<]<<      // clear s_m and go to sigma
    [>>>>[>>]<+<[<<]<<-]     // if sigma then s_m++ and sigma--
    >                        // go to mu
```

3.3 Pointer positioning

Let 0 and 1 represent moving H to the left and to the right respectively. The operation `+>>` will push the head to the right, and `<<-` will push it to the left. The idea is to move the head to the right twice if $\mu \neq 0$, then always move it to the left. It is also possible without going back to μ, which is faster, but requires the head to be sufficiently far away from C.

```
procedure MOVE_HEAD
    [>+<-]>              // move mu to Theta and go to Theta (sync)
    [->>[>>]+>>+>>]      // if Theta then Theta-- and H >> 2
```

```
    >>[>>]              // if not Theta then go to H
    <<<<[>>]           // synchronize the pointer to be at H
    <<-<<[<<](</|k+3)   // H << 1, go to Omega
```

Alternatively - the slower, but more memory-safe option:

```
procedure MOVE_HEAD
    [>>>[>>]+>>+[<<]-]  // if mu then H >> 2 and mu--
    >>>[>>]-            // H << 1
    <<[<<](</|k+3)      // go to Omega
```

3.4 Loading and storing data into C

Let T_q^0 and T_q^1 be subsequences of T_q, such that:

$$T_q^0 = [q_{01}, \ldots, q_{0k}][s_0][m_0]$$
$$T_q^1 = [q_{11}, \ldots, q_{1k}][s_1][m_1]$$

The idea behind copying the configuration is to first create a small buffer for each transition header. One of the most basic Brainfuck algorithms, the value copy algorithm (implemented as `[>+>+<<-]>>[<<+>>-]`; copies the cell value to the right), needs two cells of extra space, one of which can be reused later. For $|T_q^0| = |T_q^1| = k + 2$, the buffer would require a size of $k + 3$ cells. A procedure would first copy T_q^0 into the buffer, then if $[H] = 1$, clear the buffer and copy T_q^1 into it. This is how the headers are going to be structured:

$$T_q = [I][q_{01}, \ldots, q_{0k}][s_0][m_0][q_{11}, \ldots, q_{1k}][s_1][m_1][BUF] : BUF = [\underbrace{0, 0, \ldots, 0}_{k+3 \text{ times}}]$$

The value of τ will then be equal to $3k + 8$.

Let $C(x)$ be the procedure that copies a value under the pointer x cells to the right. For $x \in \{0, 1\}$, `FULLCOPY_x` will start and end at s_{xq} and copy T_q^x into the buffer of T_q.

```
procedure C(x)
    [(>/x)+>+(</x+1)-](>/x+1)[(</x+1)+(>/x+1)-](</x+1)
procedure FULLCOPY_0
    (C(2k+4)>/k+2)(</2k+5)
procedure FULLCOPY_1
    (C(k+2)>/k+2)(</k+3)
```

The procedures below handle pointer movement between Ω, H, and the current transition header.

```
procedure GOTO_T        // from Omega
    (</tau)[(</tau)](>/tau)
procedure COME_FROM_T   // to Omega
    [(>/tau)]
```

```
procedure GOTO_H        // from Omega
    (>|k+5)[>>]<
procedure COME_FROM_H   // to Omega
    <[<<](<|k+3)
```

The full sequence of loading the header into the buffer will be performed as follows:

```
procedure LOAD_HEADER
    GOTO_T
    >                   // go to s_0
    FULLCOPY_0
    COME_FROM_T
    GOTO_H
    [
        COME_FROM_H
        GOTO_T
        (>|2k+5)         // go to BUF
        ([-]>|k+2)       // clear BUF
        (<|2k+4)         // go to s_1
        FULLCOPY_1
        COME_FROM_T     // synchronize the position
        GOTO_H
        -               // clear [H]
    ]
    COME_FROM_H
```

Note that the $[H]$ will always be overwritten before the next position change, which is why we can clear its value while loading the header, and thus `SET_S` will not need to handle clearing $[H]$:

```
procedure SET_S
    [>>>>[>>]<+<[<<]<<-]>
```

Storing the buffer of T_q into C is simply a sequence of moving individual values and can be implemented like this:

```
procedure STORE_HEADER
    offs := 2k + 5          // BUF[0]
    GOTO_T
    for i in [0, ..., k+1] do
        (>|offs)            // go to the copied value "V"
        [                   // while V
            (>|tau-offs)     // (Omega + i)++ (store V)
            COME_FROM_T
            (>|i+1)
            +
            (<|i+1)
            GOTO_T          // V--
```

```
        (>|offs)
        -
    ]
    (<|offs)            // go back to I_T
    offs++              // point to BUF[i]
COME_FROM_T
```

3.5 Looping and halting

The easiest way of looping the program is to put the entire logic in square brackets and end the loop on a special cell that indicates whether the state is halting or not. This cell can be, for example, χ_1. If $\chi_1 \neq 0$, then $\chi \neq$ HALT, otherwise $\chi =$ HALT.

```
procedure TURING_MAIN
    INITIALIZE_STRUCTURE    // places the pointer at chi_1
    [
        -<                  // clear chi_1, go to Omega
        TURING_LOOP         // ends at Omega
        >                   // go to chi_1
    ]
    (>|k)                   // handle symbol and position changes after HALT
    SET_S < MOVE_HEAD
```

With this, `SET_Q` will skip checking χ_1, as it is only a flag and will always be set to 0 by the time this procedure is executed.

```
procedure SET_Q
    >>
    for i in [2, ..., k] do { S(i)> }
```

3.6 Initialization

Let T be the transition matrix taken from the C code example. L and R are changed to 0 and 1 respectively. All non-halting states map to the placement of their transition tuples ($trans\_0 \mapsto 1$ etc.). They are then encoded into binary numbers with uniform length $\lfloor \log_2(|Q|) \rfloor + 1$. A one is then placed at the beginning of those states (χ_1). We can then evaluate $k = \lfloor \log_2(|Q|) \rfloor + 2$ and $H \mapsto \{0\}^k$.

$$T = \begin{bmatrix} t1 & 1 & R & t1 & 1 & L \\ t0 & 1 & L & t2 & 0 & L \\ H & 1 & R & t3 & 1 & L \\ t3 & 1 & R & t0 & 0 & R \end{bmatrix} = \begin{bmatrix} 2 & 1 & 1 & 2 & 1 & 0 \\ 1 & 1 & 0 & 3 & 0 & 0 \\ 0 & 1 & 1 & 4 & 1 & 0 \\ 4 & 1 & 1 & 1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1010 & 1 & 1 & 1010 & 1 & 0 \\ 1001 & 1 & 0 & 1011 & 0 & 0 \\ 0000 & 1 & 1 & 1100 & 1 & 0 \\ 1100 & 1 & 1 & 1001 & 0 & 1 \end{bmatrix}$$

The order of the transition tuples is then reversed, and to each one, we add a single 0 at the beginning (I) and append $\{0\}^{k+3}$ (BUF) to the end, creating a

transition header space.

$$HEADERS = \begin{pmatrix} 01100111001010000000 \\ 00000111100100000000 \\ 01001101011000000000 \\ 01010111010100000000 \end{pmatrix}$$

The C section holds the entirety of T_1^0 (here: 101011), and $TAPE = 0\{01\}^p$, where p is the initial head position. All initial information can be represented by the string $HEADERS + \Omega + C + TAPE$. To put the data into memory, we map 0 to `>` and 1 to `+>`, then initialize the tape and go to χ_1 with $\langle < \rangle^{2p+k+2}$.

4 Conclusion

4.1 Synthesis

A configuration file for each Turing machine can take this form:

```
section CONFIG
    FILE [file name].bf
    HEADPOS [initial head position >= 0]
section TRANSITIONS
    /*
    Q_x - new state if [H] == x
    S_x - new symbol if [H] == x
    M_x - movement direction if [H] == x
    */
    [STATE_1] : [Q_0] [S_0] [M_0] [Q_1] [S_1] [M_1]
    [STATE_2] : [Q_0] [S_0] [M_0] [Q_1] [S_1] [M_1]
    [STATE_3] : [Q_0] [S_0] [M_0] [Q_1] [S_1] [M_1]
    ...
    [STATE_n] : [Q_0] [S_0] [M_0] [Q_1] [S_1] [M_1]
```

A compiler will then evaluate `INITIALIZE_STRUCTURE`, following the implementation steps discussed in the subsection 3.6.

```
procedure INITIALIZE_STRUCTURE
    p := HEADPOS
    INITIALIZE_HEADERS      // ends at Theta
    (>>+|p)                 // place H at p (initialize TAPE)
    (<|2p + k + 2)          // go to chi_1
```

The loop part will first go to the new state, set the new symbol, then it will move the pointer to the left/right, after which it will load and store another header into $T_{current}$:

```
procedure TURING_LOOP
    SET_Q
```

```
SET_S
MOVE_HEAD
LOAD_HEADER
STORE_HEADER
GOTO_T          // Clear all I_T values
[-(>/tau)]
```

Provided model is capable of simulating any Turing machine program, which proves that Brainfuck is Turing-complete.

□

4.2 Implementation in C++

The steps documented in section 3 allowed me to implement a Turing machine-to-Brainfuck compiler in C++. The repository can be accessed through `https://github.com/ziem24/BFT`. For more information on this project, refer to its documentation in the README file.

4.3 Remarks

The model I presented also happens to prove that Brainfuck with 1-bit values and undefined overflow/underflow behavior is Turing-complete. If the values wrap, a simple 5-instruction 1-bit Brainfuck is Turing-complete (the instructions $+$ and $-$ are effectively bit negations). Basic 5-instruction Brainfuck is Turing-complete as well, because all instances of $\langle - \rangle^n$ can be replaced with $\langle + \rangle^{256-n}$. In the particular case of this model, each minus symbol can be replaced with [+], because the only time a cell is decremented is when it is set to 0.
As for the nested conditionals, Turing-completeness can be achieved with triply nested conditions, which do not depend on the initial configuration (replacing minuses with [+] raises this number to 4).

References

`https://esolangs.org/wiki/Brainfuck`
`https://brainfuck.org/utm.b`