Máquinas de Turing para Teste de Primalidade

Vinicius Quaresma da Luz[1]

[1]Departamento de Computação – Universidade Federal de São Carlos (UFSCar)
São Carlos – SP – Brazil

viniciusluz@estudante.ufscar.br

Resumo. *Em um contexto no qual assuntos nos campos da Inteligência Artificial ganham mais tração do que nunca, é pertinente discutir o modelo teórico que possibilitou o nascimento da ideia formal de algoritmos: a Máquina de Turing. Este artigo busca discutir sobre este modelo teórico e fornecer uma implementação com viés didático utilizando a linguagem de programação Python, permitindo visualizar o funcionamento de uma máquina cujo propósito é determinar se um certo número é ou não um número primo.*

1. Introdução

A Máquina de Turing, proposta por Alan Turing em 1936, é um modelo matemático fundamental para a ciência da computação. Sua função é formalizar o conceito de algoritmo e definir os limites teóricos da computabilidade, servindo como base para o estudo do que pode ser resolvido por um processo mecânico [Turing 1937]. Apesar de sua simplicidade conceitual, seu poder computacional é universal, sendo capaz de simular qualquer algoritmo executado por computadores modernos.

Dentre os problemas clássicos da computação, o teste de primalidade ocupa um lugar de destaque. A tarefa de determinar se um número inteiro é primo ou composto é uma questão fundamental da teoria dos números, com raízes que remontam à matemática da Grécia Antiga [Knuth 1997]. Para a ciência da computação, o problema representa um campo de provas para o desenvolvimento e a análise de algoritmos, bem como para a medição de eficiência computacional.

A relevância do teste de primalidade não é apenas teórica, possuindo aplicações práticas críticas, especialmente na área de criptografia. Sistemas de chave pública, como o RSA, baseiam sua segurança na dificuldade computacional de fatorar números grandes que são o produto de dois primos [Rivest et al. 1978]. Além disso, números primos são utilizados em algoritmos de hashing e na geração de números pseudoaleatórios, tornando o problema central para diversas áreas da computação aplicada [Cormen et al. 2009].

Este trabalho tem como objetivo explorar a resolução do teste de primalidade utilizando o modelo da Máquina de Turing. O foco não reside no desenvolvimento de um algoritmo com máxima eficiência, mas na demonstração de como um modelo computacional teórico pode ser aplicado para resolver um problema concreto. Busca-se, com isso, detalhar o processo lógico e mecânico de uma computação em seu nível mais fundamental.

Para atingir este objetivo, foi projetada e implementada uma Máquina de Turing que decide sobre a primalidade de um número. A abordagem utiliza uma representação unária para a entrada e executa um algoritmo baseado em divisões sucessivas. A

implementação, realizada na linguagem Python, permite simular o modelo teórico e validar seu funcionamento, ilustrando a capacidade do modelo de Turing de encapsular e resolver problemas complexos.

2. Materiais e Métodos

2.1. A Máquina de Turing

A Máquina de Turing constitui um modelo matemático abstrato de computação que formaliza uma máquina teórica com capacidade para simular a lógica de qualquer algoritmo computável. Proposta por Alan Turing em seu artigo seminal de 1936, *"On Computable Numbers, with an Application to the Entscheidungsproblem"*, a máquina foi concebida com o propósito de formalizar o conceito de "computabilidade", bem como explorar os limites do que é mecanicamente decidível [Turing 1937]. Historicamente, o trabalho de Turing estabeleceu as fundações da ciência da computação teórica, ao prover uma definição rigorosa para a noção de algoritmo.

Formalmente, uma Máquina de Turing determinística é representada por uma 7-tupla $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, onde:

- Q é um conjunto finito de **estados** internos.
- Σ é o **alfabeto de entrada**, um conjunto finito de símbolos que não inclui o símbolo branco.
- Γ é o **alfabeto da fita**, um superconjunto finito de Σ.
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ é a **função de transição**, que dita a ação da máquina a cada passo.
- $q_0 \in Q$ é o **estado inicial**, a partir do qual a computação se inicia.
- $B \in \Gamma \setminus \Sigma$ é o **símbolo branco**, que preenche a fita além da entrada.
- $F \subseteq Q$ é o conjunto de **estados de aceitação**.

O processo computacional tem início com a cadeia de entrada disposta na fita, com a máquina no estado q_0 e o cabeçote posicionado sobre o primeiro símbolo. A cada passo, a função δ é aplicada. A computação é terminada quando a máquina atinge um estado pertencente a F, resultando na aceitação da entrada, ou quando atinge uma configuração para a qual δ não está definida, resultando na rejeição.

2.2. Técnicas para Construção de Máquinas de Turing

Embora o modelo formal da Máquina de Turing seja simples, o projeto de máquinas para resolver problemas complexos pode se tornar uma tarefa árdua. Para facilitar esse processo, foram desenvolvidas diversas técnicas conceituais que, embora não aumentem o poder computacional do modelo (pela tese de Church-Turing), simplificam o projeto e a descrição dos algoritmos. Destacam-se [Souza 2025]:

- **Armazenamento no Controle Finito:** Esta técnica permite que o estado da máquina seja representado por uma tupla de componentes, como $[q, A]$, onde q é um estado de controle e A é um símbolo armazenado na "memória" do controle finito. Isso é útil para que a máquina possa "lembrar" de um símbolo lido sem a necessidade de escrevê-lo repetidamente na fita.

- **Fitas com Múltiplas Trilhas:** A fita pode ser visualizada como se tivesse um número finito k de trilhas. Cada célula da fita contém uma tupla de k símbolos, e o cabeçote lê e escreve os k símbolos simultaneamente. Essa técnica é particularmente útil para manter diferentes informações alinhadas, como um número de entrada em uma trilha, um divisor em outra e resultados de cálculos em uma terceira, como no caso do algoritmo de teste de primalidade.
- **Marcação de Símbolos:** Consiste em utilizar um alfabeto de fita estendido para marcar símbolos que já foram processados. Frequentemente, isso é implementado com uma fita de duas trilhas, onde a primeira trilha contém a cadeia de entrada e a segunda é usada para marcar posições com um símbolo especial (e.g., *check*), indicando que o símbolo correspondente na primeira trilha já foi verificado.
- **Sub-rotinas:** Uma Máquina de Turing pode ser projetada de forma modular, onde partes do conjunto de estados são designadas para realizar tarefas específicas, funcionando como sub-rotinas. Uma sub-rotina possui um estado inicial próprio e um ou mais estados de retorno, permitindo que a máquina principal "chame" a sub-rotina para executar uma função e depois retome seu processamento.

Algumas destas técnicas são utilizadas na elaboração da máquina que é objeto deste artigo.

3. Implementação

Para a implementação da Máquina de Turing, foi utilizada a linguagem de programação Python. A estrutura adotada tenta emular o funcionamento de uma Máquina de Turing, gerenciando elementos de seu estado como a posição do controle interno, estado das fitas, dentre outros. O programa aceita uma entrada binária $w_1 \geq 2_{(10)}$ e fornece um resultado $r \in \{q_{\text{accept}}, q_{\text{reject}}\}$.

O código-fonte completo da máquina de Turing em Python, discutido neste artigo, está disponível publicamente no repositório GitHub abaixo.

- **Repositório GitHub:** `https://github.com/seu-usuario/seu-repositorio`

Abaixo, buscando elucidar como o código funciona e melhor explicar o processo de testes na seção subsequente, estão alguns trechos adaptados do código.

```
class TuringMachineDefinition:
    Q: Set[State]
    Sigma: Set[str]
    Gamma: Set[str]
    delta: Delta
    q0: State
    BlankSymbol: str
    F: Set[State]

    def __init__(
        self,
        Q: Set[State],
        Sigma: Set[str],
        Gamma: Set[str],
        delta: Delta,
        q0: State,
```

```
        BlankSymbol: str,
        F: Set[State],
    ):
        self.Q = Q
        self.Sigma = Sigma
        self.Gamma = Gamma
        self.delta = delta
        self.q0 = q0
        self.BlankSymbol = BlankSymbol
        self.F = F
```

Código 1. Definição da Máquina de Turing

A classe que consome essa definição é a classe TURINGMACHINE, exposta abaixo. Note que algumas omissões foram feitas para fins de brevidade.

```
class TuringMachine:
    _definition: TuringMachineDefinition
    _tape: Dict[int, TapeBlock]
    _head_position: int
    _current_state: State
    _verbose: bool
    _step_count: int

    def __init__(
        self,
        definition: TuringMachineDefinition,
        w_1: str,
        w_2: Optional[str] = None,
        w_3: Optional[str] = None,
        verbose: bool = False,
    ):
        self._verbose = verbose
        self._definition = definition
        self._current_state = self._definition.q0
        self._head_position = 0
        self._step_count = 0
        # ...
        self._tape = {
            i: (w_1[i], w_2[i], w_3[i]) for i in range(max_len)
        }

    def run(self) -> TuringMachineExcecutionResult:
        while self._current_state not in self._definition.F:
            self.step()
        # ...
        return TuringMachineExcecutionResult(
            step_count=self._step_count,
            final_state=self._current_state,
        )

    def step(self):
        self._step_count += 1
        # ...
        next_state, write_template, direction = self._fd_transition()
        current_symbols = self.current_tape_block
        new_symbols_to_write = tuple(
```

```
            current_symbols[i]
                if write_template[i] == "*"
                else write_template[i] for i in range(3)
        )
        # ...
        self._current_state = next_state
        self._tape[self._head_position] = new_symbols_to_write
        self._head_position += direction.value
    # ...
```

Código 2. Executor para a definição da Máquina de Turing

A lógica principal da simulação ocorre no método STEP e no método FIND_TRANSITION. No método STEP, o programa avalia seu estado atual e o conteúdo da fita sob o controle da Máquina de Turing. Com isso, ele avalia a função δ e determina qual será o próximo passo. A função FIND_TRANSITION é a responsável por encontrar o δ, ou lançar um MISSINGTRANSITIONERROR caso não encontre. Caso a função lance a exceção, a máquina tem o seu funcionamento interrompido.

Um ponto digno de menção é que esta implementação oferece suporte a *wildcards* nas regras de definição de δ, simplificando expressivamente a definição das regras.

```
def _find_transition(self) -> Tuple[State, TapeBlock, Direction]:
    current_symbols = self.current_tape_block

    for (
        state,
        read_symbols,
    ), transition_output in self._definition.delta.items():
        if state == self._current_state:
            is_match = all(
                read_symbols[i] == current_symbols[i]
                or read_symbols[i] == "*"
                for i in range(3)
            )
            if is_match:
                return transition_output

    raise MissingTransitionError(
        f"Não existe \delta({self._current_state}, {current_symbols})"
    )
```

Código 3. Método para encontrar δ com base no estado atual da máquina

Além disso, a cada passo da máquina, caso a opção *verbose* esteja habilitada, ela exibe em sua saída padrão as informações no seguinte formato:

```
->          delta(q_seek_end, ('1', '1', '0'))
Sub-rotina: compare_ge
F1:         | B | 1 | 0 | 1 | 1 | B | B ...
F2:         | B | 0 | 0 | 1 | 1 | B | B ...
F3:         | B | 1 | 0 | 0 | 0 | B | B ...
CTRL:                   ^   compare_ge__q_seek_end

Próximo estado: q_seek_end (Parte da sub-rotina: compare_ge)
Escrever:       ('1', '1', '0')
```

```
Direção:          RIGHT
```

Código 4. Informações exibidas pelo método step

A linha CTRL mostra uma seta apontando para onde está o controle da máquina naquele instante, e isso é atualizado visualmente a cada iteração da máquina.

3.1. Algoritmo utilizado

```
ALGORITMO TestePrimalidade
INICIO
    FITA_1 <- ENTRADA

    # Dividir
    FITA_2 <- (2)_10

    # Loop principal, responsável por aumentar o divisor
    # Fica em execução até aceitar, rejeitar ou até a máquina parar.
    ENQUANTO VERDADEIRO FAÇA
        SE FITA_1 = FITA_2 ENTÃO
            ACEITA
            FIMALGORITMO
        SENÃO
            FITA_3 <- FITA_1

            # Loop de subtração
            ENQUANTO FITA_3 >= FITA_2 FAÇA
                FITA_3 <- FITA_3 - FITA_2
            FIMENQUANTO

            SE FITA_3 = (0)_10 ENTÃO
                REJEITA
                FIMALGORITMO
            SENÃO
                FITA_2 <- FITA_2 + (1)_10
            FIMSE
        FIMSE
    FIMENQUANTO
FIMALGORITMO
```

Código 5. Pseudocódigo utilizado para guiar o projeto e implementação da Máquina de Turing

3.2. Definição da Máquina de Turing

Utilizando as demais classes definidas para a execução deste projeto, foram criadas as definições e regras necessárias para expressar o algoritmo desejado utilizando a máquina de Turing.

Para criar esta máquina, foi adotada a estratégia de quebrá-la em sub-rotinas. A notação utilizada foi a de que o nome da sub-rotina é seguido por 2 *underlines* e então pelo nome do estado. Isto é utilizado pela visualização completa posteriormente para facilitar a compreensão de qual sub-rotina está em execução a cada passo.

Abaixo está sua definição. Por conta de sua extensão, a definição de δ é exibida em uma figura separada logo abaixo.

```
primality_test_definition = TuringMachineDefinition(
    Q={
        # Estados do Orquestrador
        "q_start",
        "q_accept",
        "q_reject",
        # Sub-rotina: INIT_DIVISOR
        "init__q0",
        "init__q_write0",
        "init__q_write1",
        "init__q_fill_zeros",
        "init__q_rewind",
        # Sub-rotina: COMPARE_EQUALITY (N == D)
        "compare_eq__q_seek_end",
        "compare_eq__q_compare",
        "compare_eq__q_notequal_rewind",
        # Sub-rotina: COPY_TAPE_1_TO_3
        "copy__q_copy",
        "copy__q_rewind",
        # Sub-rotina: COMPARE_GREATER_EQUAL (T3 >= T2)
        "compare_ge__q_seek_end",
        "compare_ge__q_compare_len",
        "compare_ge__q_rewind_msb",
        "compare_ge__q_compare_msb",
        "compare_ge__q_rewind_false",
        "compare_ge__q_rewind_true",
        # Sub-rotina: SUBTRACT
        "subtract__q0",
        "subtract__q1",
        "subtract__q2",
        "subtract__q_rewind",
        # Sub-rotina: CHECK_REMAINDER_IS_ZERO
        "check_remainder__q_check",
        "check_remainder__q_rewind",
        # Sub-rotina: CLEANUP_TAPE_3
        "cleanup__q_0",
        "cleanup__q_rewind",
        # Sub-rotina: INCREMENT_DIVISOR
        "increment__q_seek_end",
        "increment__q_add_carry",
        "increment__q_rewind",
    },
    Sigma={"0", "1"},
    Gamma={"0", "1", "B"},
    BlankSymbol="B",
    q0="q_start",
    F={"q_accept", "q_reject"},
    delta=# Incluído na figura abaixo.
)
```

Código 6. Definição da Máquina de Turing para teste de primalidade

```
# ========================================
# Ponto de Entrada
# ========================================
("q_start", ("*", "*", "*")): (
    "init__q0",
```

```
    ("*", "*", "*"),
    Direction.RIGHT,
),
# ========================================
# SUB-ROTINA 1: INIT_DIVISOR (FITA_2 <- 2)
# ========================================
("init__q0", ("0", "*", "*")): (
    "init__q0",
    ("*", "0", "*"),
    Direction.RIGHT,
),
("init__q0", ("1", "*", "*")): (
    "init__q0",
    ("*", "0", "*"),
    Direction.RIGHT,
),
("init__q0", ("B", "B", "B")): (
    "init__q_write0",
    ("B", "B", "B"),
    Direction.LEFT,
),
("init__q_write0", ("*", "*", "*")): (
    "init__q_write1",
    ("*", "0", "*"),
    Direction.LEFT,
),
("init__q_write1", ("*", "*", "*")): (
    "init__q_fill_zeros",
    ("*", "1", "*"),
    Direction.LEFT,
),
("init__q_fill_zeros", ("0", "*", "*")): (
    "init__q_fill_zeros",
    ("*", "0", "*"),
    Direction.LEFT,
),
("init__q_fill_zeros", ("1", "*", "*")): (
    "init__q_fill_zeros",
    ("*", "0", "*"),
    Direction.LEFT,
),
("init__q_fill_zeros", ("B", "B", "B")): (
    "init__q_rewind",
    ("B", "B", "B"),
    Direction.RIGHT,
),
("init__q_rewind", ("0", "*", "*")): (
    "init__q_rewind",
    ("*", "*", "*"),
    Direction.LEFT,
),
("init__q_rewind", ("1", "*", "*")): (
    "init__q_rewind",
    ("*", "*", "*"),
    Direction.LEFT,
),
```

```
("init__q_rewind", ("B", "B", "B")): (
    "compare_eq__q_seek_end",
    ("B", "B", "B"),
    Direction.RIGHT,
),
# ========================================
# SUB-ROTINA 2: COMPARE_EQUALITY (SE FITA_1 == FITA_2)
# ========================================
("compare_eq__q_seek_end", ("0", "*", "*")): (
    "compare_eq__q_seek_end",
    ("0", "*", "*"),
    Direction.RIGHT,
),
("compare_eq__q_seek_end", ("1", "*", "*")): (
    "compare_eq__q_seek_end",
    ("1", "*", "*"),
    Direction.RIGHT,
),
("compare_eq__q_seek_end", ("B", "B", "B")): (
    "compare_eq__q_compare",
    ("B", "B", "B"),
    Direction.LEFT,
),
("compare_eq__q_compare", ("0", "0", "*")): (
    "compare_eq__q_compare",
    ("0", "0", "*"),
    Direction.LEFT,
),
("compare_eq__q_compare", ("1", "1", "*")): (
    "compare_eq__q_compare",
    ("1", "1", "*"),
    Direction.LEFT,
),
("compare_eq__q_compare", ("B", "B", "B")): (
    "q_accept",
    ("B", "B", "B"),
    Direction.RIGHT,
),
("compare_eq__q_compare", ("0", "1", "*")): (
    "compare_eq__q_notequal_rewind",
    ("*", "*", "*"),
    Direction.LEFT,
),
("compare_eq__q_compare", ("1", "0", "*")): (
    "compare_eq__q_notequal_rewind",
    ("*", "*", "*"),
    Direction.LEFT,
),
("compare_eq__q_notequal_rewind", ("0", "*", "*")): (
    "compare_eq__q_notequal_rewind",
    ("*", "*", "*"),
    Direction.LEFT,
),
("compare_eq__q_notequal_rewind", ("1", "*", "*")): (
    "compare_eq__q_notequal_rewind",
    ("*", "*", "*"),
```

```
    Direction.LEFT,
),
("compare_eq__q_notequal_rewind", ("B", "B", "B")): (
    "copy__q_copy",
    ("B", "B", "B"),
    Direction.RIGHT,
),
# ========================================
# SUB-ROTINA 3: COPY_TAPE_1_TO_3 (FITA_3 <- FITA_1)
# ========================================
("copy__q_copy", ("0", "*", "*")): (
    "copy__q_copy",
    ("*", "*", "0"),
    Direction.RIGHT,
),
("copy__q_copy", ("1", "*", "*")): (
    "copy__q_copy",
    ("*", "*", "1"),
    Direction.RIGHT,
),
("copy__q_copy", ("B", "B", "B")): (
    "copy__q_rewind",
    ("B", "B", "B"),
    Direction.LEFT,
),
("copy__q_rewind", ("0", "*", "*")): (
    "copy__q_rewind",
    ("*", "*", "*"),
    Direction.LEFT,
),
("copy__q_rewind", ("1", "*", "*")): (
    "copy__q_rewind",
    ("*", "*", "*"),
    Direction.LEFT,
),
("copy__q_rewind", ("B", "B", "B")): (
    "compare_ge__q_seek_end",
    ("B", "B", "B"),
    Direction.RIGHT,
),
# ========================================
# SUB-ROTINA 4: COMPARE_GREATER_EQUAL (SE FITA_3 >= FITA_2)
# ========================================
("compare_ge__q_seek_end", ("0", "*", "*")): (
    "compare_ge__q_seek_end",
    ("0", "*", "*"),
    Direction.RIGHT,
),
("compare_ge__q_seek_end", ("1", "*", "*")): (
    "compare_ge__q_seek_end",
    ("1", "*", "*"),
    Direction.RIGHT,
),
("compare_ge__q_seek_end", ("B", "B", "B")): (
    "compare_ge__q_compare_len",
    ("B", "B", "B"),
```

```
    Direction.LEFT,
),
("compare_ge__q_compare_len", ("*", "0", "0")): (
    "compare_ge__q_compare_len",
    ("*", "0", "0"),
    Direction.LEFT,
),
("compare_ge__q_compare_len", ("*", "1", "1")): (
    "compare_ge__q_compare_len",
    ("*", "1", "1"),
    Direction.LEFT,
),
("compare_ge__q_compare_len", ("*", "0", "1")): (
    "compare_ge__q_compare_len",
    ("*", "0", "1"),
    Direction.LEFT,
),
("compare_ge__q_compare_len", ("*", "1", "0")): (
    "compare_ge__q_compare_len",
    ("*", "1", "0"),
    Direction.LEFT,
),
("compare_ge__q_compare_len", ("*", "B", "0")): (
    "compare_ge__q_rewind_true",
    ("*", "B", "0"),
    Direction.LEFT,
),
("compare_ge__q_compare_len", ("*", "B", "1")): (
    "compare_ge__q_rewind_true",
    ("*", "B", "1"),
    Direction.LEFT,
),
("compare_ge__q_compare_len", ("*", "0", "B")): (
    "compare_ge__q_rewind_false",
    ("*", "0", "B"),
    Direction.LEFT,
),
("compare_ge__q_compare_len", ("*", "1", "B")): (
    "compare_ge__q_rewind_false",
    ("*", "1", "B"),
    Direction.LEFT,
),
("compare_ge__q_compare_len", ("B", "B", "B")): (
    "compare_ge__q_rewind_msb",
    ("B", "B", "B"),
    Direction.RIGHT,
),
("compare_ge__q_rewind_msb", ("0", "*", "*")): (
    "compare_ge__q_rewind_msb",
    ("0", "*", "*"),
    Direction.LEFT,
),
("compare_ge__q_rewind_msb", ("1", "*", "*")): (
    "compare_ge__q_rewind_msb",
    ("1", "*", "*"),
    Direction.LEFT,
```

```
),
("compare_ge__q_rewind_msb", ("B", "B", "B")): (
    "compare_ge__q_compare_msb",
    ("B", "B", "B"),
    Direction.RIGHT,
),
("compare_ge__q_compare_msb", ("*", "0", "0")): (
    "compare_ge__q_compare_msb",
    ("*", "0", "0"),
    Direction.RIGHT,
),
("compare_ge__q_compare_msb", ("*", "1", "1")): (
    "compare_ge__q_compare_msb",
    ("*", "1", "1"),
    Direction.RIGHT,
),
("compare_ge__q_compare_msb", ("*", "0", "1")): (
    "compare_ge__q_rewind_true",
    ("*", "0", "1"),
    Direction.LEFT,
),
("compare_ge__q_compare_msb", ("*", "1", "0")): (
    "compare_ge__q_rewind_false",
    ("*", "1", "0"),
    Direction.LEFT,
),
("compare_ge__q_compare_msb", ("B", "B", "B")): (
    "compare_ge__q_rewind_true",
    ("B", "B", "B"),
    Direction.LEFT,
),
("compare_ge__q_rewind_true", ("0", "*", "*")): (
    "compare_ge__q_rewind_true",
    ("*", "*", "*"),
    Direction.LEFT,
),
("compare_ge__q_rewind_true", ("1", "*", "*")): (
    "compare_ge__q_rewind_true",
    ("*", "*", "*"),
    Direction.LEFT,
),
("compare_ge__q_rewind_true", ("B", "B", "B")): (
    "subtract__q0",
    ("B", "B", "B"),
    Direction.RIGHT,
),
("compare_ge__q_rewind_false", ("0", "*", "*")): (
    "compare_ge__q_rewind_false",
    ("*", "*", "*"),
    Direction.LEFT,
),
("compare_ge__q_rewind_false", ("1", "*", "*")): (
    "compare_ge__q_rewind_false",
    ("*", "*", "*"),
    Direction.LEFT,
),
```

```
("compare_ge__q_rewind_false", ("B", "B", "B")): (
    "check_remainder__q_check",
    ("B", "B", "B"),
    Direction.RIGHT,
),
# ========================================
# SUB-ROTINA 5: SUBTRACT (FITA_3 <- FITA_3 - FITA_2)
# ========================================
("subtract__q0", ("0", "*", "*")): (
    "subtract__q0",
    ("*", "*", "*"),
    Direction.RIGHT,
),
("subtract__q0", ("1", "*", "*")): (
    "subtract__q0",
    ("*", "*", "*"),
    Direction.RIGHT,
),
("subtract__q0", ("B", "B", "B")): (
    "subtract__q1",
    ("B", "B", "B"),
    Direction.LEFT,
),
("subtract__q1", ("*", "0", "0")): (
    "subtract__q1",
    ("*", "*", "0"),
    Direction.LEFT,
),
("subtract__q1", ("*", "0", "1")): (
    "subtract__q1",
    ("*", "*", "1"),
    Direction.LEFT,
),
("subtract__q1", ("*", "1", "1")): (
    "subtract__q1",
    ("*", "*", "0"),
    Direction.LEFT,
),
("subtract__q1", ("*", "1", "0")): (
    "subtract__q2",
    ("*", "*", "1"),
    Direction.LEFT,
),
("subtract__q2", ("*", "0", "1")): (
    "subtract__q1",
    ("*", "*", "0"),
    Direction.LEFT,
),
("subtract__q2", ("*", "0", "0")): (
    "subtract__q2",
    ("*", "*", "1"),
    Direction.LEFT,
),
("subtract__q2", ("*", "1", "1")): (
    "subtract__q2",
    ("*", "*", "1"),
```

```
    Direction.LEFT,
),
("subtract__q2", ("*", "1", "0")): (
    "subtract__q2",
    ("*", "*", "0"),
    Direction.LEFT,
),
("subtract__q1", ("B", "B", "B")): (
    "subtract__q_rewind",
    ("B", "B", "B"),
    Direction.RIGHT,
),
("subtract__q2", ("B", "B", "B")): (
    "subtract__q_rewind",
    ("B", "B", "B"),
    Direction.RIGHT,
),
("subtract__q_rewind", ("0", "*", "*")): (
    "subtract__q_rewind",
    ("*", "*", "*"),
    Direction.LEFT,
),
("subtract__q_rewind", ("1", "*", "*")): (
    "subtract__q_rewind",
    ("*", "*", "*"),
    Direction.LEFT,
),
("subtract__q_rewind", ("B", "B", "B")): (
    "compare_ge__q_seek_end",
    ("B", "B", "B"),
    Direction.RIGHT,
),
# ========================================
# SUB-ROTINA 6: CHECK_REMAINDER_IS_ZERO (SE FITA_3 == 0)
# ========================================
("check_remainder__q_check", ("*", "*", "0")): (
    "check_remainder__q_check",
    ("*", "*", "*"),
    Direction.RIGHT,
),
("check_remainder__q_check", ("*", "*", "1")): (
    "check_remainder__q_rewind",
    ("*", "*", "*"),
    Direction.LEFT,
),
("check_remainder__q_check", ("B", "B", "B")): (
    "q_reject",
    ("B", "B", "B"),
    Direction.RIGHT,
),
("check_remainder__q_rewind", ("0", "*", "*")): (
    "check_remainder__q_rewind",
    ("*", "*", "*"),
    Direction.LEFT,
),
("check_remainder__q_rewind", ("1", "*", "*")): (
```

```
    "check_remainder__q_rewind",
    ("*", "*", "*"),
    Direction.LEFT,
),
("check_remainder__q_rewind", ("B", "B", "B")): (
    "cleanup__q_0",
    ("B", "B", "B"),
    Direction.RIGHT,
),
# ========================================
# SUB-ROTINA 7: CLEANUP_TAPE_3 (Limpa Fita 3)
# ========================================
("cleanup__q_0", ("*", "*", "0")): (
    "cleanup__q_0",
    ("*", "*", "B"),
    Direction.RIGHT,
),
("cleanup__q_0", ("*", "*", "1")): (
    "cleanup__q_0",
    ("*", "*", "B"),
    Direction.RIGHT,
),
("cleanup__q_0", ("B", "B", "B")): (
    "cleanup__q_rewind",
    ("B", "B", "B"),
    Direction.LEFT,
),
("cleanup__q_rewind", ("0", "*", "*")): (
    "cleanup__q_rewind",
    ("*", "*", "*"),
    Direction.LEFT,
),
("cleanup__q_rewind", ("1", "*", "*")): (
    "cleanup__q_rewind",
    ("*", "*", "*"),
    Direction.LEFT,
),
("cleanup__q_rewind", ("B", "B", "B")): (
    "increment__q_seek_end",
    ("B", "B", "B"),
    Direction.RIGHT,
),
# ========================================
# SUB-ROTINA 8: INCREMENT_DIVISOR (FITA_2 <- FITA_2 + 1)
# ========================================
("increment__q_seek_end", ("0", "*", "*")): (
    "increment__q_seek_end",
    ("*", "*", "*"),
    Direction.RIGHT,
),
("increment__q_seek_end", ("1", "*", "*")): (
    "increment__q_seek_end",
    ("*", "*", "*"),
    Direction.RIGHT,
),
("increment__q_seek_end", ("B", "B", "B")): (
```

```
454     "increment__q_add_carry",
455     ("B", "B", "B"),
456     Direction.LEFT,
457 ),
458 ("increment__q_add_carry", ("*", "0", "*")): (
459     "increment__q_rewind",
460     ("*", "1", "*"),
461     Direction.LEFT,
462 ),
463 ("increment__q_add_carry", ("*", "1", "*")): (
464     "increment__q_add_carry",
465     ("*", "0", "*"),
466     Direction.LEFT,
467 ),
468 ("increment__q_add_carry", ("*", "B", "*")): (
469     "increment__q_rewind",
470     ("*", "1", "*"),
471     Direction.LEFT,
472 ),
473 ("increment__q_rewind", ("0", "*", "*")): (
474     "increment__q_rewind",
475     ("*", "*", "*"),
476     Direction.LEFT,
477 ),
478 ("increment__q_rewind", ("1", "*", "*")): (
479     "increment__q_rewind",
480     ("*", "*", "*"),
481     Direction.LEFT,
482 ),
483 ("increment__q_rewind", ("B", "B", "B")): (
484     "compare_eq__q_seek_end",
485     ("B", "B", "B"),
486     Direction.RIGHT,
487 )
```

Código 7. Definição da função δ da Máquina de Turing para teste de primalidade

Nestas próximas subseções, as sub-rotinas são explicadas em maior detalhe. Junto a cada uma delas, é disponibilizada a definição da função δ para aquela sub-rotina em questão. Note que a máquina objeto deste relatório é composta pelo funcionamento em conjunto de todas estas sub-rotinas.

3.2.1. Sub-rotina de inicialização

A sub-rotina de inicialização copia o valor 2_{10} para a fita 2. Isso é o ponto de partida para que possamos, nos próximos passos, ir incrementando o divisor. Além disso, essa e as demais sub-rotinas possuem sempre uma **etapa de rebobinação**, para que o controle seja devolvido para a próxima sub-rotina sempre estando na posição de índice zero. Por isso observamos o estado q_{rewind}, que volta a fita até a extremidade esquerda e então passa o fluxo de controle para a sub-rotina seguinte.

$$\delta(\text{init\_\_q0}, (0, *, *)) = (\text{init\_\_q0}, (*, 0, *), R)$$

$$\delta(\text{init\_\_q0}, (1, *, *)) = (\text{init\_\_q0}, (*, 0, *), R)$$

$$\delta(\text{init\_\_q0}, (B, B, B)) = (\text{init\_\_q\_write0}, (B, B, B), L)$$

$$\delta(\text{init\_\_q\_write0}, (*, *, *)) = (\text{init\_\_q\_write1}, (*, 0, *), L)$$

$$\delta(\text{init\_\_q\_write1}, (*, *, *)) = (\text{init\_\_q\_fill\_zeros}, (*, 1, *), L)$$

$$\delta(\text{init\_\_q\_fill\_zeros}, (0, *, *)) = (\text{init\_\_q\_fill\_zeros}, (*, 0, *), L)$$

$$\delta(\text{init\_\_q\_fill\_zeros}, (1, *, *)) = (\text{init\_\_q\_fill\_zeros}, (*, 0, *), L)$$

$$\delta(\text{init\_\_q\_fill\_zeros}, (B, B, B)) = (\text{init\_\_q\_rewind}, (B, B, B), R)$$

$$\delta(\text{init\_\_q\_rewind}, (0, *, *)) = (\text{init\_\_q\_rewind}, (*, *, *), L)$$

$$\delta(\text{init\_\_q\_rewind}, (1, *, *)) = (\text{init\_\_q\_rewind}, (*, *, *), L)$$

$$\delta(\text{init\_\_q\_rewind}, (B, B, B)) = (\text{compare\_eq\_\_q\_seek\_end}, (B, B, B), R)$$

3.2.2. Sub-rotina de comparação de igualdade

A próxima sub-rotina fica responsável por comparar a fita 1 com a fita 2. Se o divisor (fita 2) for igual ao número de entrada (fita 1), significa que o número não foi divisível por nenhum valor menor que ele, caracterizando-o como primo. Nesse caso, a máquina entra no estado de aceitação. Caso contrário, o fluxo segue para a próxima sub-rotina.

$$\delta(\text{compare\_eq\_\_q\_seek\_end}, (0, *, *)) = (\text{compare\_eq\_\_q\_seek\_end}, (0, *, *), R)$$

$$\delta(\text{compare\_eq\_\_q\_seek\_end}, (1, *, *)) = (\text{compare\_eq\_\_q\_seek\_end}, (1, *, *), R)$$

$$\delta(\text{compare\_eq\_\_q\_seek\_end}, (B, B, B)) = (\text{compare\_eq\_\_q\_compare}, (B, B, B), L)$$

$$\delta(\text{compare\_eq\_\_q\_compare}, (0, 0, *)) = (\text{compare\_eq\_\_q\_compare}, (0, 0, *), L)$$

$$\delta(\text{compare\_eq\_\_q\_compare}, (1, 1, *)) = (\text{compare\_eq\_\_q\_compare}, (1, 1, *), L)$$

$$\delta(\text{compare\_eq\_\_q\_compare}, (B, B, B)) = (\text{q\_accept}, (B, B, B), R)$$

$$\delta(\text{compare\_eq\_\_q\_compare}, (0, 1, *)) = (\text{compare\_eq\_\_q\_notequal\_rewind}, (*, *, *), L)$$

$$\delta(\text{compare\_eq\_\_q\_compare}, (1, 0, *)) = (\text{compare\_eq\_\_q\_notequal\_rewind}, (*, *, *), L)$$

$$\delta(\text{compare\_eq\_\_q\_notequal\_rewind}, (0, *, *)) = (\text{compare\_eq\_\_q\_notequal\_rewind}, (*, *, *), L)$$

$$\delta(\text{compare\_eq\_\_q\_notequal\_rewind}, (1, *, *)) = (\text{compare\_eq\_\_q\_notequal\_rewind}, (*, *, *), L)$$

$$\delta(\text{compare\_eq\_\_q\_notequal\_rewind}, (B, B, B)) = (\text{copy\_\_q\_copy}, (B, B, B), R)$$

3.2.3. Sub-rotina de cópia

A sub-rotina de cópia é responsável por transcrever o conteúdo da Fita 1 para a Fita 3. Este passo é crucial para preservar o valor original do dividendo (o número de entrada), permitindo que as operações de subtração sejam realizadas em uma cópia, sem alterar o valor que será testado nas iterações seguintes.

$$\delta(\text{copy\_\_q\_copy}, (0, *, *)) = (\text{copy\_\_q\_copy}, (*, *, 0), R)$$
$$\delta(\text{copy\_\_q\_copy}, (1, *, *)) = (\text{copy\_\_q\_copy}, (*, *, 1), R)$$
$$\delta(\text{copy\_\_q\_copy}, (B, B, B)) = (\text{copy\_\_q\_rewind}, (B, B, B), L)$$
$$\delta(\text{copy\_\_q\_rewind}, (0, *, *)) = (\text{copy\_\_q\_rewind}, (*, *, *), L)$$
$$\delta(\text{copy\_\_q\_rewind}, (1, *, *)) = (\text{copy\_\_q\_rewind}, (*, *, *), L)$$
$$\delta(\text{copy\_\_q\_rewind}, (B, B, B)) = (\text{compare\_ge\_\_q\_seek\_end}, (B, B, B), R)$$

3.2.4. Sub-rotina de comparação "maior ou igual que"

Esta sub-rotina implementa a comparação $T_3 \geq T_2$. Ela compara os bits do mais significativo para o menos significativo. O resultado desta comparação determina se o loop de subtrações sucessivas deve continuar. Se $T_3 < T_2$, a divisão pelo divisor atual termina.

$$\delta(\text{compare\_ge\_\_q\_seek\_end}, (0, *, *)) = (\text{compare\_ge\_\_q\_seek\_end}, (0, *, *), R)$$
$$\delta(\text{compare\_ge\_\_q\_seek\_end}, (1, *, *)) = (\text{compare\_ge\_\_q\_seek\_end}, (1, *, *), R)$$
$$\delta(\text{compare\_ge\_\_q\_seek\_end}, (B, B, B)) = (\text{compare\_ge\_\_q\_compare\_len}, (B, B, B), L)$$
$$\delta(\text{compare\_ge\_\_q\_compare\_len}, (*, 0, 0)) = (\text{compare\_ge\_\_q\_compare\_len}, (*, 0, 0), L)$$
$$\delta(\text{compare\_ge\_\_q\_compare\_len}, (*, 1, 1)) = (\text{compare\_ge\_\_q\_compare\_len}, (*, 1, 1), L)$$
$$\delta(\text{compare\_ge\_\_q\_compare\_len}, (*, 0, 1)) = (\text{compare\_ge\_\_q\_compare\_len}, (*, 0, 1), L)$$
$$\delta(\text{compare\_ge\_\_q\_compare\_len}, (*, 1, 0)) = (\text{compare\_ge\_\_q\_compare\_len}, (*, 1, 0), L)$$
$$\delta(\text{compare\_ge\_\_q\_compare\_len}, (*, B, 0)) = (\text{compare\_ge\_\_q\_rewind\_true}, (*, B, 0), L)$$
$$\delta(\text{compare\_ge\_\_q\_compare\_len}, (*, B, 1)) = (\text{compare\_ge\_\_q\_rewind\_true}, (*, B, 1), L)$$
$$\delta(\text{compare\_ge\_\_q\_compare\_len}, (*, 0, B)) = (\text{compare\_ge\_\_q\_rewind\_false}, (*, 0, B), L)$$
$$\delta(\text{compare\_ge\_\_q\_compare\_len}, (*, 1, B)) = (\text{compare\_ge\_\_q\_rewind\_false}, (*, 1, B), L)$$
$$\delta(\text{compare\_ge\_\_q\_compare\_len}, (B, B, B)) = (\text{compare\_ge\_\_q\_rewind\_msb}, (B, B, B), R)$$
$$\delta(\text{compare\_ge\_\_q\_rewind\_msb}, (0, *, *)) = (\text{compare\_ge\_\_q\_rewind\_msb}, (0, *, *), L)$$
$$\delta(\text{compare\_ge\_\_q\_rewind\_msb}, (1, *, *)) = (\text{compare\_ge\_\_q\_rewind\_msb}, (1, *, *), L)$$
$$\delta(\text{compare\_ge\_\_q\_rewind\_msb}, (B, B, B)) = (\text{compare\_ge\_\_q\_compare\_msb}, (B, B, B), R)$$
$$\delta(\text{compare\_ge\_\_q\_compare\_msb}, (*, 0, 0)) = (\text{compare\_ge\_\_q\_compare\_msb}, (*, 0, 0), R)$$

$$\delta(\text{compare\_ge\_\_q\_compare\_msb}, (*, 1, 1)) = (\text{compare\_ge\_\_q\_compare\_msb}, (*, 1, 1), R)$$
$$\delta(\text{compare\_ge\_\_q\_compare\_msb}, (*, 0, 1)) = (\text{compare\_ge\_\_q\_rewind\_true}, (*, 0, 1), L)$$
$$\delta(\text{compare\_ge\_\_q\_compare\_msb}, (*, 1, 0)) = (\text{compare\_ge\_\_q\_rewind\_false}, (*, 1, 0), L)$$
$$\delta(\text{compare\_ge\_\_q\_compare\_msb}, (B, B, B)) = (\text{compare\_ge\_\_q\_rewind\_true}, (B, B, B), L)$$
$$\delta(\text{compare\_ge\_\_q\_rewind\_true}, (0, *, *)) = (\text{compare\_ge\_\_q\_rewind\_true}, (*, *, *), L)$$
$$\delta(\text{compare\_ge\_\_q\_rewind\_true}, (1, *, *)) = (\text{compare\_ge\_\_q\_rewind\_true}, (*, *, *), L)$$
$$\delta(\text{compare\_ge\_\_q\_rewind\_true}, (B, B, B)) = (\text{subtract\_\_q0}, (B, B, B), R)$$
$$\delta(\text{compare\_ge\_\_q\_rewind\_false}, (0, *, *)) = (\text{compare\_ge\_\_q\_rewind\_false}, (*, *, *), L)$$
$$\delta(\text{compare\_ge\_\_q\_rewind\_false}, (1, *, *)) = (\text{compare\_ge\_\_q\_rewind\_false}, (*, *, *), L)$$
$$\delta(\text{compare\_ge\_\_q\_rewind\_false}, (B, B, B)) = (\text{check\_remainder\_\_q\_check}, (B, B, B), R)$$

3.2.5. Sub-rotina de subtração

A sub-rotina de subtração realiza a operação $T_3 \leftarrow T_3 - T_2$ utilizando o método de complemento de dois. Este é o núcleo do laço de divisão, sendo executada repetidamente enquanto o valor na Fita 3 for maior ou igual ao da Fita 2.

$$\delta(\text{subtract\_\_q0}, (0, *, *)) = (\text{subtract\_\_q0}, (*, *, *), R)$$
$$\delta(\text{subtract\_\_q0}, (1, *, *)) = (\text{subtract\_\_q0}, (*, *, *), R)$$
$$\delta(\text{subtract\_\_q0}, (B, B, B)) = (\text{subtract\_\_q1}, (B, B, B), L)$$
$$\delta(\text{subtract\_\_q1}, (*, 0, 0)) = (\text{subtract\_\_q1}, (*, *, 0), L)$$
$$\delta(\text{subtract\_\_q1}, (*, 0, 1)) = (\text{subtract\_\_q1}, (*, *, 1), L)$$
$$\delta(\text{subtract\_\_q1}, (*, 1, 1)) = (\text{subtract\_\_q1}, (*, *, 0), L)$$
$$\delta(\text{subtract\_\_q1}, (*, 1, 0)) = (\text{subtract\_\_q2}, (*, *, 1), L)$$
$$\delta(\text{subtract\_\_q2}, (*, 0, 1)) = (\text{subtract\_\_q1}, (*, *, 0), L)$$
$$\delta(\text{subtract\_\_q2}, (*, 0, 0)) = (\text{subtract\_\_q2}, (*, *, 1), L)$$
$$\delta(\text{subtract\_\_q2}, (*, 1, 1)) = (\text{subtract\_\_q2}, (*, *, 1), L)$$
$$\delta(\text{subtract\_\_q2}, (*, 1, 0)) = (\text{subtract\_\_q2}, (*, *, 0), L)$$
$$\delta(\text{subtract\_\_q1}, (B, B, B)) = (\text{subtract\_\_q\_rewind}, (B, B, B), R)$$
$$\delta(\text{subtract\_\_q2}, (B, B, B)) = (\text{subtract\_\_q\_rewind}, (B, B, B), R)$$
$$\delta(\text{subtract\_\_q\_rewind}, (0, *, *)) = (\text{subtract\_\_q\_rewind}, (*, *, *), L)$$
$$\delta(\text{subtract\_\_q\_rewind}, (1, *, *)) = (\text{subtract\_\_q\_rewind}, (*, *, *), L)$$
$$\delta(\text{subtract\_\_q\_rewind}, (B, B, B)) = (\text{compare\_ge\_\_q\_seek\_end}, (B, B, B), R)$$

3.2.6. Sub-rotina de verificação do resto

Após a conclusão do loop de subtrações, esta sub-rotina verifica se o valor restante na Fita 3 é zero. Para isso, ela percorre a fita e, se encontrar apenas símbolos '0', conclui que a divisão foi exata. Neste caso, o número de entrada não é primo, e a máquina transita para o estado de rejeição.

$$\delta(\text{check\_remainder\_\_q\_check}, (*, *, 0)) = (\text{check\_remainder\_\_q\_check}, (*, *, *), R)$$

$$\delta(\text{check\_remainder\_\_q\_check}, (*, *, 1)) = (\text{check\_remainder\_\_q\_rewind}, (*, *, *), L)$$

$$\delta(\text{check\_remainder\_\_q\_check}, (B, B, B)) = (\text{q\_reject}, (B, B, B), R)$$

$$\delta(\text{check\_remainder\_\_q\_rewind}, (0, *, *)) = (\text{check\_remainder\_\_q\_rewind}, (*, *, *), L)$$

$$\delta(\text{check\_remainder\_\_q\_rewind}, (1, *, *)) = (\text{check\_remainder\_\_q\_rewind}, (*, *, *), L)$$

$$\delta(\text{check\_remainder\_\_q\_rewind}, (B, B, B)) = (\text{cleanup\_\_q\_0}, (B, B, B), R)$$

3.2.7. Sub-rotina de limpeza

A sub-rotina de limpeza é executada quando a divisão não é exata. Sua única função é apagar o conteúdo da Fita 3, preenchendo-a com o símbolo branco. Isso prepara a fita para a próxima iteração do loop principal, onde ela receberá uma nova cópia do número de entrada.

$$\delta(\text{cleanup\_\_q\_0}, (*, *, 0)) = (\text{cleanup\_\_q\_0}, (*, *, B), R)$$

$$\delta(\text{cleanup\_\_q\_0}, (*, *, 1)) = (\text{cleanup\_\_q\_0}, (*, *, B), R)$$

$$\delta(\text{cleanup\_\_q\_0}, (B, B, B)) = (\text{cleanup\_\_q\_rewind}, (B, B, B), L)$$

$$\delta(\text{cleanup\_\_q\_rewind}, (0, *, *)) = (\text{cleanup\_\_q\_rewind}, (*, *, *), L)$$

$$\delta(\text{cleanup\_\_q\_rewind}, (1, *, *)) = (\text{cleanup\_\_q\_rewind}, (*, *, *), L)$$

$$\delta(\text{cleanup\_\_q\_rewind}, (B, B, B)) = (\text{increment\_\_q\_seek\_end}, (B, B, B), R)$$

3.2.8. Sub-rotina de incremento

Por fim, a sub-rotina de incremento adiciona 1 ao valor do divisor na Fita 2. Após o incremento, o controle retorna para a sub-rotina de comparação de igualdade, reiniciando o ciclo de verificação com o próximo divisor.

$$\delta(\text{increment\_\_q\_seek\_end}, (0, *, *)) = (\text{increment\_\_q\_seek\_end}, (*, *, *), R)$$

$$\delta(\text{increment\_\_q\_seek\_end}, (1, *, *)) = (\text{increment\_\_q\_seek\_end}, (*, *, *), R)$$

$$\delta(\text{increment\_\_q\_seek\_end}, (B, B, B)) = (\text{increment\_\_q\_add\_carry}, (B, B, B), L)$$

$$\delta(\text{increment\_\_q\_add\_carry}, (*, 0, *)) = (\text{increment\_\_q\_rewind}, (*, 1, *), L)$$
$$\delta(\text{increment\_\_q\_add\_carry}, (*, 1, *)) = (\text{increment\_\_q\_add\_carry}, (*, 0, *), L)$$
$$\delta(\text{increment\_\_q\_add\_carry}, (*, B, *)) = (\text{increment\_\_q\_rewind}, (*, 1, *), L)$$
$$\delta(\text{increment\_\_q\_rewind}, (0, *, *)) = (\text{increment\_\_q\_rewind}, (*, *, *), L)$$
$$\delta(\text{increment\_\_q\_rewind}, (1, *, *)) = (\text{increment\_\_q\_rewind}, (*, *, *), L)$$
$$\delta(\text{increment\_\_q\_rewind}, (B, B, B)) = (\text{compare\_eq\_\_q\_seek\_end}, (B, B, B), R)$$

4. Testes

A máquina foi submetida a um conjunto de valores primos conhecidos e a um conjunto de números que sabe-se não terem esta propriedade. A cada execução, foi mensurado o tempo de execução e a quantidade de passos tomada pela máquina. Além disso, os resultados foram comparados com a função IS_PRIME_FN, que é uma função simples para determinar se um valor é ou não primo.

Nos testes, disponíveis por completo no repositório da implementação [da Luz 2025], a Máquina de Turing ofereceu uma performance muito pior. Todavia, em nenhum dos cenários de teste, sua saída diferiu da função IS_PRIME_FN. Isso mostra que o comportamento da Máquina de Turing está alinhado com as expectativas, ao menos detre os valores que foram testados.

A fim de exemplificação, abaixo está o final do resultado da execução para o valor 7_{10}.

```
A máquina encerrou sua execução.
Estado final: q_accept
Conteúdo da fita 1: 111
Conteúdo da fita 2: 111
Conteúdo da fita 3: (vazio)
Posição da cabeça: 0
Contagem de passos: 456
==| Resultado da Máquina de Turing |==
Resultado: q_accept, Passos: 456
Tempo gasto: 0.011638 segundos

Para fins de comparação, a função is_prime_fn também será executada.

==| Resultado da função is_prime_fn |==
Resultado da função: True, Passos: 2
Tempo gasto na função: 0.000010 segundos

==| Comparação de Desempenho |==
A máquina de Turing levou 0.011638 segundos, enquanto a função levou
    0.000010 segundos.
A máquina de Turing foi 116280.00% mais lenta que a função.
A máquina de Turing executou 456 passos, enquanto a função executou 2
    passos.
==| Fim da execução |==
```

Código 8. Resultado para a entrada 7

Como outro exemplo, aqui está o resultado para 281_{10}.

```
A máquina encerrou sua execução.
Estado final: q_accept
Conteúdo da fita 1: 100011001
Conteúdo da fita 2: 100011001
Conteúdo da fita 3: (vazio)
Posição da cabeça: 0
Contagem de passos: 98850
==| Resultado da Máquina de Turing |==
Resultado: q_accept, Passos: 98850
Tempo gasto: 0.985857 segundos

Para fins de comparação, a função is_prime_fn também será executada.

==| Resultado da função is_prime_fn |==
Resultado da função: True, Passos: 16
Tempo gasto na função: 0.000009 segundos

==| Comparação de Desempenho |==
A máquina de Turing levou 0.985857 segundos, enquanto a função levou
    0.000009 segundos.
A máquina de Turing foi 10953866.67% mais lenta que a função.
A máquina de Turing executou 98850 passos, enquanto a função executou
    16 passos.
==| Fim da execução |==
```

Código 9. Resultado para a entrada 281

5. Conclusão

Este artigo mostrou o processo de implementação de uma Máquina de Turing em Python para avaliar se um determinado valor é ou não primo. Os testes conduzidos ao final mostraram o correto comportamento da máquina, e mecanismos visuais foram implantadados para tornar seu resultado mais didático e fácil de compreender.

Em termos de performance, o comportamento da Máquina de Turing foi muito inferior ao da função simples, que já possuia as primitivas necessárias para testar a primalidade de maneira simples. Todavia, isso não descarta de maneira alguma a importância deste modelo teórico, que segue sendo extremamente relevante para a computação mesmo depois de quase um século de sua concepção.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. The MIT Press.

da Luz, V. Q. (2025). Implementação de uma máquina de turing em python. Repositório de código. Disponível em: `https://github.com/seu-usuario/seu-repositorio`.

Knuth, D. E. (1997). *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley Professional.

Rivest, R. L., Shamir, A., and Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126.

Souza, W. L. d. (2025). Aula 16: Máquinas de turing (mt) - 9.1 definição formal de mt. Material de aula, Disciplina de Teoria da Computação.

Turing, A. M. (1937). On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265.