Instituto Tecnológico de Aeronáutica - ITA
CT-213 - Inteligência Artificial aplicada à Robótica Móvel
Aluno: Ulisses Lopes da Silva

Relatório do Laboratório 1 - Máquina de Estados Finita e *Behavior Tree*

---

1 Breve Explicação em Alto Nível da Implementação

A implementação do comportamento do robô *Roomba* seguiu as diretrizes estabelecidas pelo código-base, utilizando a propriedade do polimorfismo, própria da programação orientada a objetos.

Com relação ao código base, foi acrescentada em `constants.py` a importação da biblioteca `math`, a fim de utilizar π como a constante **PI**. Além disso, para melhor visualização do comportamento do robô, foi aumentada a lista de pontos para 4000 no arquivo `simulation.py`. Dessa forma, o rastro deixado pelo robô era maior e perdurava por mais tempo na tela, melhorando a visualização do seu comportamento.

1.1 Máquina de Estados Finita

Para a Máquina de Estados Finita, um contador `self.counter` foi incluído no construtor de todas as classes, a fim de monitorar o tempo decorrido em cada comportamento, ao invocar o método `check_transition()`, e, também, para definir algumas ações que dependiam do tempo, como nas classes `MoveInSpiral()` e `Rotate()`. Nos métodos `execute()` foram implementados essencialmente apenas os comportamentos/ações dos estados. A seguir, uma explicação básica de cada estado implementado:

- `MoveForwardState()`:

  O método `check_transition()` obtém o estado de colisão por meio do método `get_bumper_state()`. Se colidiu, muda o estado para `GoBackState()`. Se não colidiu, verifica se o tempo decorrido é maior que o permitido para `MoveForwardState()`. Se for maior, muda o estado para `MoveInSpiralState()`.

  O método `execute()` executa o comportamento do estado, mudando a velocidade linear com `set_velocity()` e incrementando o contador.

- `MoveInSpiralState()`:

  O método `check_transition()` obtém o estado de colisão por meio do método `get_bumper_state()`. Se colidiu, muda o estado para `GoBackState()`. Se não colidiu, verifica se o

tempo decorrido é maior que o permitido para `MoveInSpiralState()`. Se for maior, muda o estado para `MoveForwardState()`.

O método `execute()` executa o comportamento do estado, obtendo, tempo a tempo, o raio naquele instante por meio da equação fornecida e a respectiva velocidade angular consequente, mudando as velocidades linear e angular com `set_velocity()` e incrementando o contador.

As equações utilizadas foram:

$$R(t) = R_o + b(nS) \tag{1}$$
$$w(t) = v_o/R(t) \tag{2}$$

onde **n** é o contador e **S** é o *Sample Time*

- `GoBackState()`:

  O método `check_transition()` verifica se o tempo decorrido é maior que o permitido para `GoBackState()`. Se for maior, muda o estado para `RotateState()`.

  O método `execute()` executa o comportamento do estado, mudando a velocidade linear com `set_velocity()` (desta vez, com velocidade negativa da biblioteca de constantes, para ele se mover para trás) e incrementando o contador.

- `RotateState()`:

  O construtor da classe reinicia o contador e obtém um ângulo aleatório. O método `check_transition()` verifica se o ângulo percorrido é maior que o obtido aleatoriamente. Se for maior, muda o estado para `MoveForwardState()`.

  O método `execute()` executa o comportamento do estado, mudando a velocidade angular com `set_velocity()` e incrementando o contador.

  A equação utilizada foi:

$$\omega nS = \theta \Rightarrow nS = \theta/\omega \tag{3}$$

1.2 *Behavior Tree*

Para a *Behavior Tree*, primeiramente foi construída uma árvore por meio da classe `Roomba BehaviorTree`, criando a raiz como um *Selector* e as sub-árvores "Normal behavior"e "Collision behavior"(ambas com raiz do tipo *Sequence*). Em seguida, elas foram unidas entre si por meio do método `add_child()`.

Nas classes dos nós, no método `enter()` foi implementado um contador `self.counter` para monitorar o tempo de execução dos nós, ou seja, o tempo de `RUNNING`. Além disso, especificamente no nó `RotateNode()`, esse método também contava com a instância de um valor de ângulo aleatório. A seguir, uma explicação básica de cada nó implementado:

- `MoveForwardNode()`:

  O método `execute()` obtém o estado de colisão por meio do método `get_bumper_state()`. Se houve colisão, o nó retorna `FAILURE`. Se não colidiu, verifica se o tempo decorrido é maior que o permitido para `MoveForwardNode()`. Se for maior, retorna `SUCCESS`, pois completou com sucesso o ciclo. Se o tempo decorrido não for maior, modifica a velocidade linear por meio do método `set_velocity()`, incrementa o contador e retorna `RUNNING` para continuar rodando o nó.

- `MoveInSpiralNode()`:

  O método `execute()` obtém o estado de colisão por meio do método `get_bumper_state()`. Se houve colisão, o nó retorna `FAILURE`. Se não colidiu, verifica se o tempo decorrido é maior que o permitido para `MoveInSpiralNode()`. Se for maior, retorna `SUCCESS`, pois completou com sucesso o ciclo. Se o tempo decorrido não for maior, modifica as velocidades linear e angular por meio do método `set_velocity()` (utilizando as mesmas equações (1) e (2) mencionadas na MEF), incrementa o contador e retorna `RUNNING` para continuar rodando o nó.

- `GoBackNode()`:

  O método `execute()` verifica se o tempo decorrido é maior que o permitido para `GoBack Node()`. Se for maior, retorna `SUCCESS`, pois terminou o ciclo com sucesso. Se não for maior, o método `execute()` executa o comportamento do estado, mudando a velocidade linear com `set_velocity()` (desta vez, com a velocidade negativa da biblioteca de constantes, para ele se mover para trás), incrementa o contador e retorna `RUNNING`, para continuar rodando o nó.

- `RotateNode()`:

  O construtor da classe reinicia o contador e obtém um ângulo aleatório. O método `execute()` verifica se o ângulo percorrido é maior que o obtido aleatoriamente. Se for maior, retorna `SUCCESS`, pois terminou o ciclo com sucesso. Se não for maior, o método `execute()` executa o comportamento do estado, mudando a velocidade angular com `set_velocity()`, incrementa o contador e retorna `RUNNING`, para continuar rodando o nó. A equação utilizada é a (3), também mencionada na MEF.

2 Figuras Comprovando Funcionamento do Código

2.1 Máquina de Estados Finita

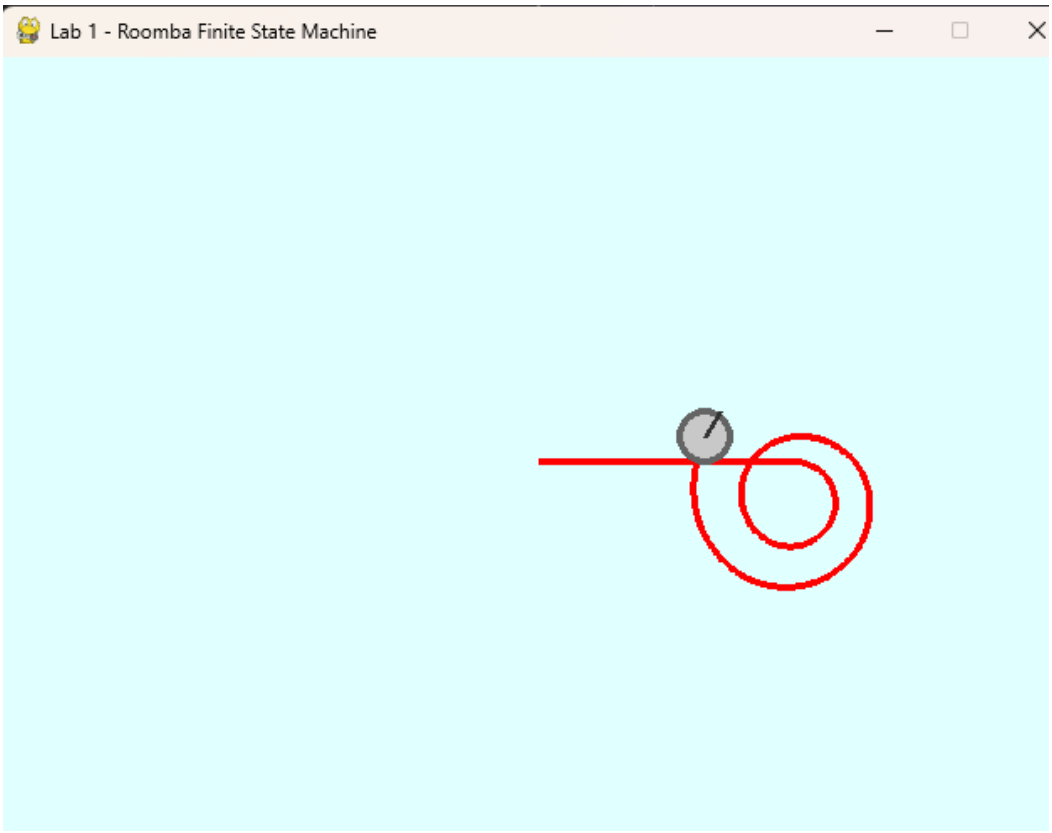

Fig. 1: Percurso do robô *Roomba* - Máquina de Estados Finita

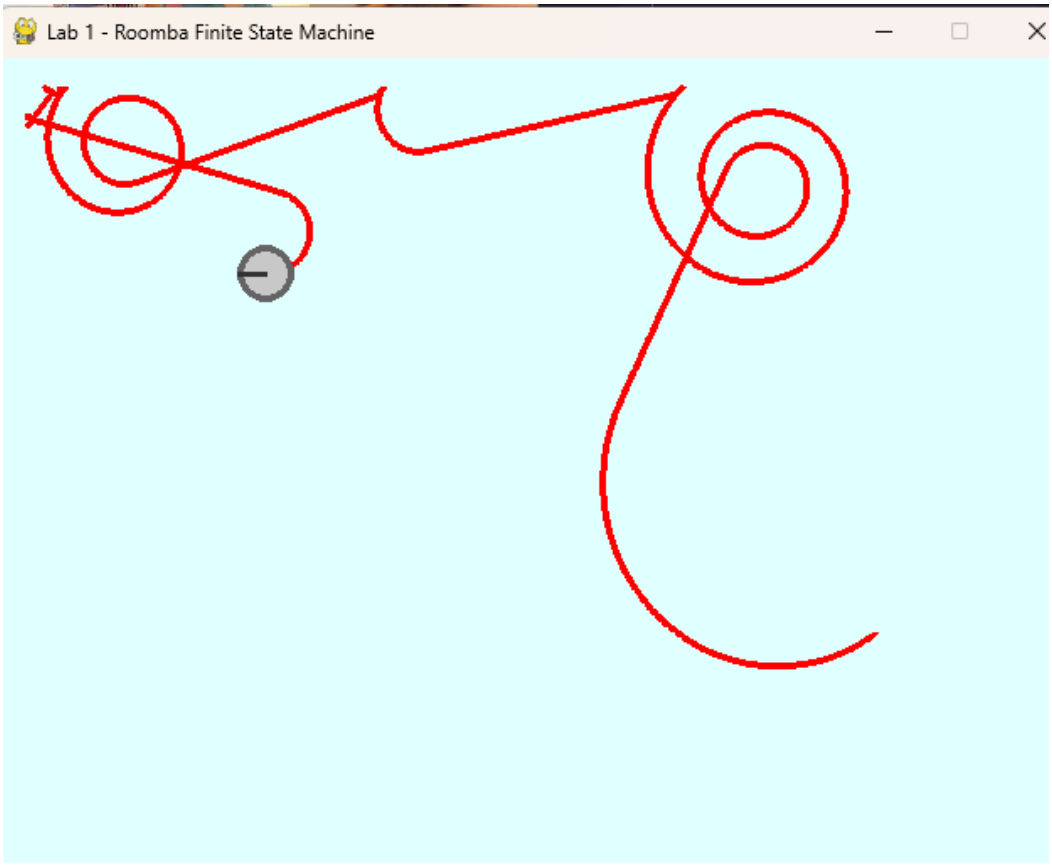

Fig. 2: Percurso do robô *Roomba* - Máquina de Estados Finita

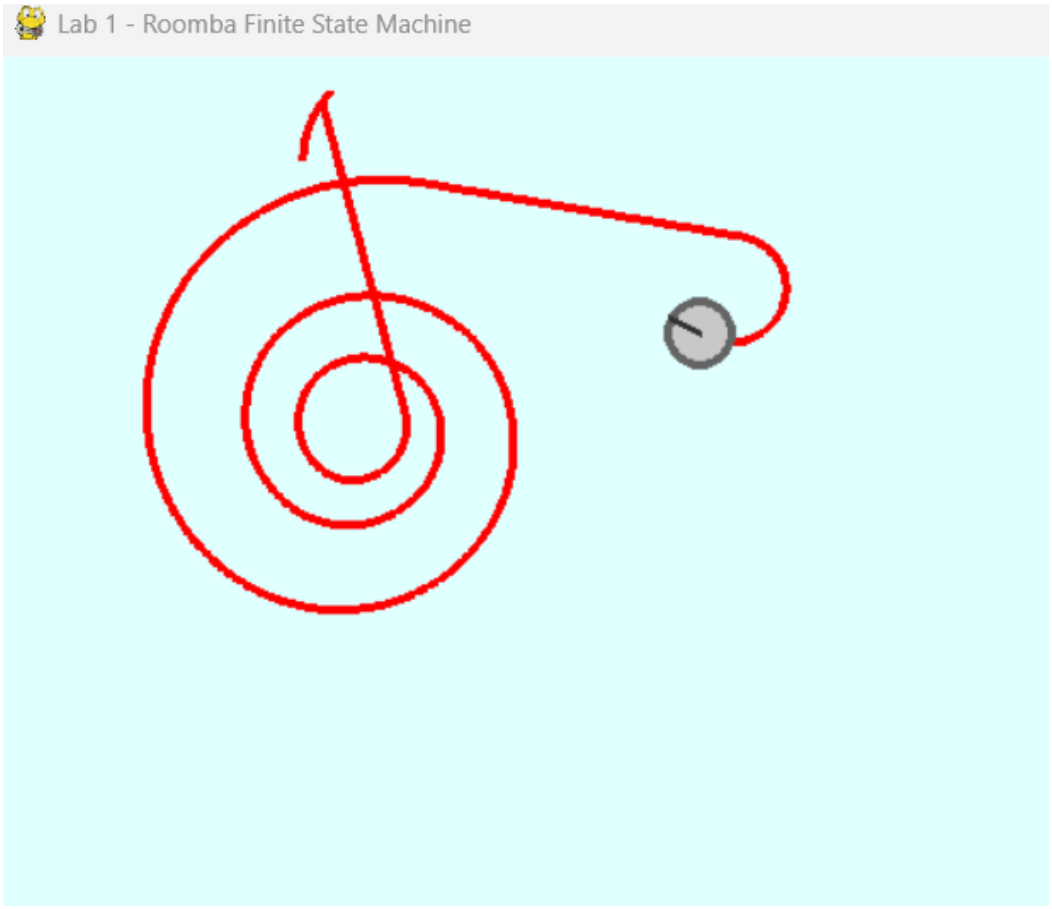

Fig. 3: Percurso do robô *Roomba* - Máquina de Estados Finita

2.2 *Behavior Tree*

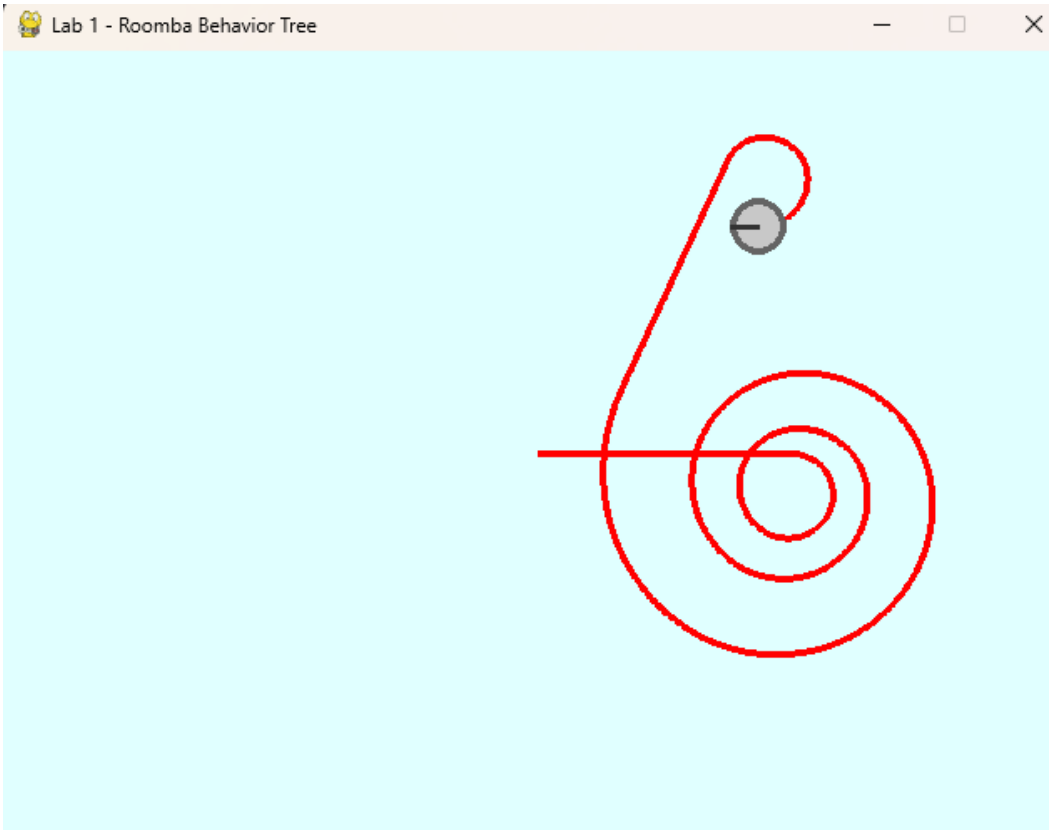

Fig. 4: Percurso do robô *Roomba - Behavior Tree*

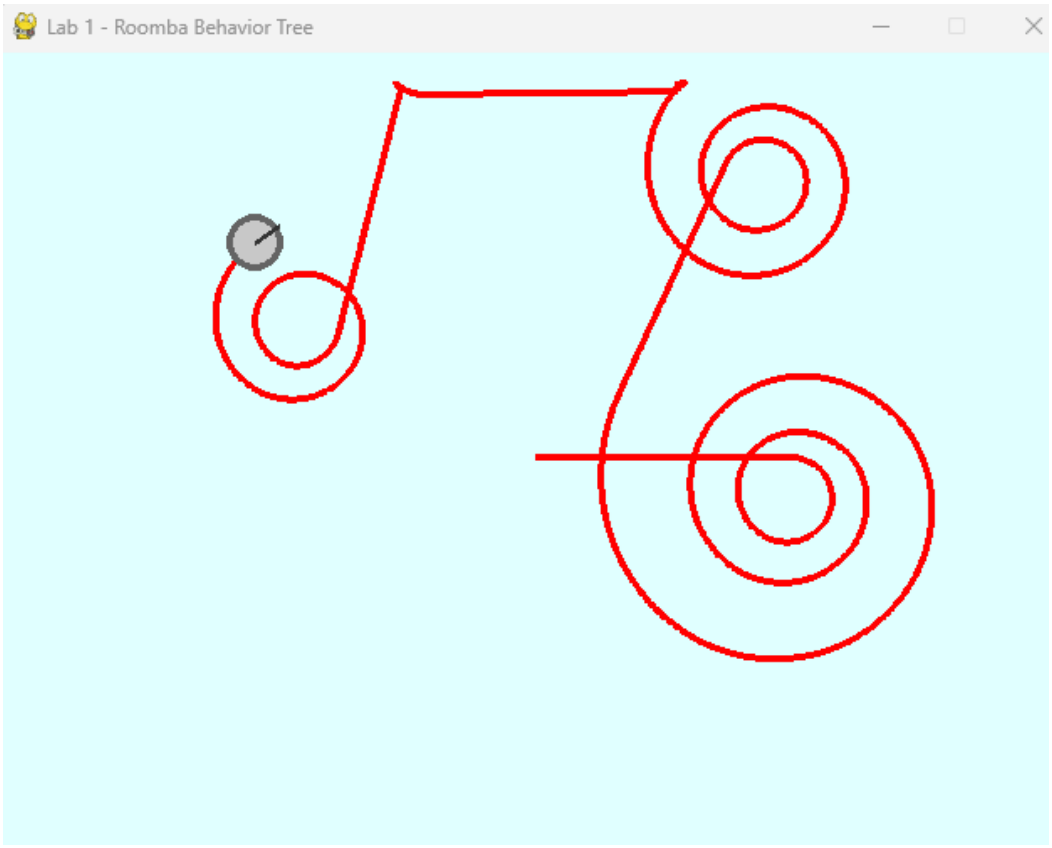

Fig. 5: Percurso do robô *Roomba - Behavior Tree*

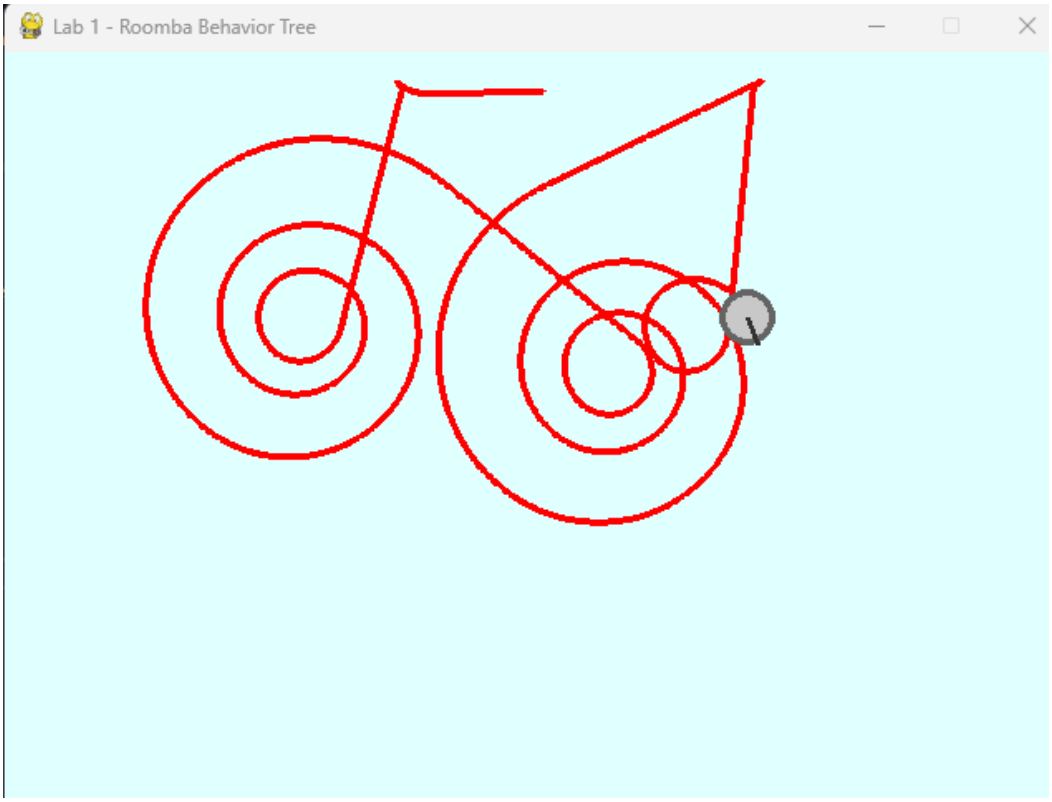

Fig. 6: Percurso do robô *Roomba - Behavior Tree*

3 APÊNDICE

Os códigos com as respectivas implementações seguem abaixo.

- Máquina de Estados Finita

```
    class MoveForwardState(State):
    def __init__(self):
        super().__init__("MoveForward")
        # Todo: add initialization code

        # Initializing the counter
        self.counter = 0

    def check_transition(self, agent, state_machine):
        # Todo: add logic to check and execute state transition

        # Checking if the roomba collided
        collided = agent.get_bumper_state()

        if collided:
            state_machine.change_state(GoBackState())
        else:
            # If did not collide, verify the elapsed time
            if (self.counter * SAMPLE_TIME) > MOVE_FORWARD_TIME:
                state_machine.change_state(MoveInSpiralState())

    def execute(self, agent):
        # Todo: add execution logic

        # State behavior
        agent.set_velocity(FORWARD_SPEED, 0)

        # incrementing the counter to calculate the elapsed time
        self.counter += 1

class MoveInSpiralState(State):
    def __init__(self):
        super().__init__("MoveInSpiral")
        # Todo: add initialization code

        # Initializing the counter
        self.counter = 0

    def check_transition(self, agent, state_machine):
        # Todo: add logic to check and execute state transition

        # Checking if the roomba collided
        collided = agent.get_bumper_state()

        if collided:
            state_machine.change_state(GoBackState())
        else:
```

```
            # If did not collide, verify the elapsed time
            if (self.counter * SAMPLE_TIME) > MOVE_IN_SPIRAL_TIME:
                state_machine.change_state(MoveForwardState())

    def execute(self, agent):
        # Todo: add execution logic

        # R(t) = Ro + b * t
        r_t = INITIAL_RADIUS_SPIRAL + SPIRAL_FACTOR * (self.counter *
            SAMPLE_TIME)

        # w(t) = v / R(t)
        w_t = FORWARD_SPEED / (r_t)

        # State behavior
        agent.set_velocity(FORWARD_SPEED, w_t)

        # incrementing the counter to calculate the elapsed time
        self.counter += 1


class GoBackState(State):
    def __init__(self):
        super().__init__("GoBack")
        # Todo: add initialization code

        # Initializing the counter
        self.counter = 0

    def check_transition(self, agent, state_machine):
        # Todo: add logic to check and execute state transition

        # checking the time moving back
        if (self.counter * SAMPLE_TIME) > GO_BACK_TIME:
            state_machine.change_state(RotateState())

    def execute(self, agent):
        # Todo: add execution logic

        # State behavior
        agent.set_velocity(BACKWARD_SPEED, 0)

        # incrementing the counter to calculate the elapsed time
        self.counter += 1


class RotateState(State):
    def __init__(self):
        super().__init__("Rotate")
        # Todo: add initialization code

        # Initializing the counter
```

```
        self.counter = 0

        # Select a random value in [-PI, PI)
        self.rotate_angle = random.uniform(-PI, PI - 1e-10)

    def check_transition(self, agent, state_machine):
        # Todo: add logic to check and execute state transition
        if (self.counter * SAMPLE_TIME) > abs(self.rotate_angle /
            ANGULAR_SPEED):
             state_machine.change_state(MoveForwardState())

    def execute(self, agent):
        # Todo: add execution logic

        # State behavior
        if self.rotate_angle < 0:
             agent.set_velocity(0, -ANGULAR_SPEED)
        else:
             agent.set_velocity(0, ANGULAR_SPEED)

        # incrementing the counter to calculate the elapsed time
        self.counter += 1
```

Listing 1: Código do Programa em Python

- *Behavior Tree*

```
    class RoombaBehaviorTree(BehaviorTree):
    """
    Represents a behavior tree of a roomba cleaning robot.
    """
    def __init__(self):
        super().__init__()
        # Todo: construct the tree here

        # Creating the tree nodes

        # Tree root
        root = SelectorNode("RootIsASelector")

        # Normal behavior sub-tree
        normal_behavior = SequenceNode("NormalBehavior")
        normal_behavior.add_child(MoveForwardNode())
        normal_behavior.add_child(MoveInSpiralNode())


        # Collision behavior sub-tree
        collision_behavior = SequenceNode("CollisionBehavior")
        collision_behavior.add_child(GoBackNode())
        collision_behavior.add_child(RotateNode())

        # Constructing the tree
        root.add_child(normal_behavior)
```

```
        root.add_child(collision_behavior)

        self.root = root


class MoveForwardNode(LeafNode):
    def __init__(self):
        super().__init__("MoveForward")
        # Todo: add initialization code

    def enter(self, agent):
        # Todo: add enter logic

        # Initializing the counter
        self.counter = 0

    def execute(self, agent):
        # Todo: add execution logic

        # Checking if the roomba collided
        collided = agent.get_bumper_state()

        if collided:
            return ExecutionStatus.FAILURE
        else:
            # If did not collide, verify the elapsed time
            if (self.counter * SAMPLE_TIME) > MOVE_FORWARD_TIME:
                return ExecutionStatus.SUCCESS
            else:
                # node task
                agent.set_velocity(FORWARD_SPEED, 0)

                # incrementing the counter to calculate the elapsed time
                self.counter += 1
                return ExecutionStatus.RUNNING


class MoveInSpiralNode(LeafNode):
    def __init__(self):
        super().__init__("MoveInSpiral")
        # Todo: add initialization code

    def enter(self, agent):
        # Todo: add enter logic

        # Initializing the counter
        self.counter = 0

    def execute(self, agent):
        # Todo: add execution logic

        # Checking if the roomba collided
```

```
        collided = agent.get_bumper_state()

        if collided:
            return ExecutionStatus.FAILURE
        else:
            # If did not collide, verify the elapsed time
            if (self.counter * SAMPLE_TIME) > MOVE_IN_SPIRAL_TIME:
                return ExecutionStatus.SUCCESS
            else:
                # R(t) = Ro + b * t
                r_t = INITIAL_RADIUS_SPIRAL + SPIRAL_FACTOR * (self.
                    counter * SAMPLE_TIME)

                # w(t) = v / R(t)
                w_t = FORWARD_SPEED / (r_t)

                # # node task
                agent.set_velocity(FORWARD_SPEED, w_t)

                # incrementing the counter to calculate the elapsed time
                self.counter += 1

                return ExecutionStatus.RUNNING


class GoBackNode(LeafNode):
    def __init__(self):
        super().__init__("GoBack")
        # Todo: add initialization code

    def enter(self, agent):
        # Todo: add enter logic

        # Initializing the counter
        self.counter = 0

    def execute(self, agent):
        # Todo: add execution logic

        # checking the time moving back
        if (self.counter * SAMPLE_TIME) > GO_BACK_TIME:
            return ExecutionStatus.SUCCESS
        else:
            # # node task
            agent.set_velocity(BACKWARD_SPEED, 0)

            # incrementing the counter to calculate the elapsed time
            self.counter += 1

            return ExecutionStatus.RUNNING
```

```python
class RotateNode(LeafNode):
    def __init__(self):
        super().__init__("Rotate")
        # Todo: add initialization code

    def enter(self, agent):
        # Todo: add enter logic

        # Initializing the counter
        self.counter = 0

        # Select a random value in [-PI, PI)
        self.rotate_angle = random.uniform(-PI, PI - 1e-10)

    def execute(self, agent):
        # Todo: add execution logic

        # Checking the rotate angle
        if (self.counter * SAMPLE_TIME) > abs(self.rotate_angle /
            ANGULAR_SPEED):
            return ExecutionStatus.SUCCESS
        else:

            # # node task
            if self.rotate_angle < 0:
                agent.set_velocity(0, -ANGULAR_SPEED)
            else:
                agent.set_velocity(0, ANGULAR_SPEED)

            # incrementing the counter to calculate the elapsed time
            self.counter += 1

            return ExecutionStatus.RUNNING
```

Listing 2: Código do Programa em Python