

## Lab 01:

O Lab foi desenvolvido com sucesso. Dividi a explicação relativa à máquina de estados na parte de state machine e a relativa a Behavior Tree na parte Behavior Tree, cada uma com imagens e trechos de código.

O código alterado está em anexo junto a esse relatório.

Gostaria de fazer algumas sugestões ao professor. Geralmente em engines, não se fixa o movimento dos objetos a um frame rate fixo, pois diferentes dispositivos têm diferentes performance, então nem todos os dispositivos irão conseguir rodar 60 fps como foi fixado na simulação. Dessa forma, o movimento do robô, ao invés de depender de um `dt = SAMPLE_TIME = 1.0 / FREQUENCY`, deveria depender de `pygame.time.Clock().get_fps()`, portanto `SAMPLE_TIME` deveria ser `SAMPLE_TIME = 1.0 / self.clock.get_fps()`, de forma que `clock` foi definido no construtor e a cada frame é chamada a função `self.clock.tick()`. O código já com essas alterações está no código em anexo.

Fazendo isso toda a movimentação do jogo fica independente do fps, permitindo tanto fps maiores que 60 quanto menores e, portanto, faz com que o jogo rode igualmente em diferentes computadores.

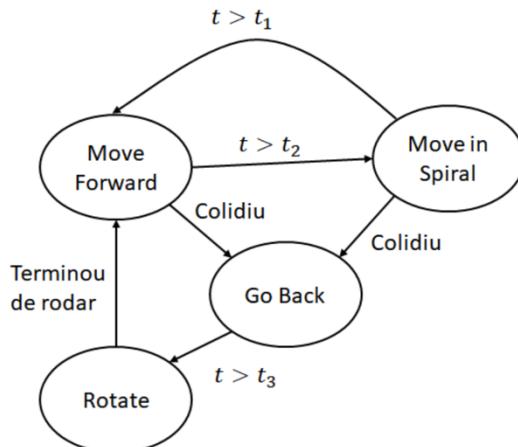
Digo isso principalmente porque estou usando MacOs e o pygame possui problema de otimização com o esse sistema operacional. Então existem momentos de queda de frame rate o que faz com que, devido a maneira que estava implementada a movimentação, o robô tivesse um comportamento de andar mais devagar ou mais rápido dependendo do frame rate que estivesse na hora da simulação.

Tive alguns problemas compatibilidade do pygame com o python 3.7 utilizando Macos, se mais alguém tiver esses problemas avise que se for utilizado o python 3.6 ele deve funcionar.

*State Machine* – De maneira simplificada, é possível reduzir o funcionamento de todas as classes dos estados implementadas à função execute e à função check\_transition. Em execute serão setados os parâmetros no agente para que ele execute a ação do estado corretamente, enquanto que no check\_transition será verificado se determinada condição para mudança de estado é atendida e se for, o estado mudará.

A construção dos estados foi feita de acordo com o Esquema 1.

Todos as classes que representam os estados da máquina de estados finitos foram completadas com sucesso e seus códigos estão definidos do Código 1 ao Código 4. Os códigos Código 5 e Código 6 se referem a funções auxiliares criadas e usada nas classes dos estados. As imagens mostradas da Figura 1 à Figura 5 denotam o comportamento do robô bem como suas transições de estados.



Esquema 1: Esquema da máquina de estados finitos

```

class MoveForwardState(State):
    def __init__(self):
        super().__init__("MoveForward")
        self.initial_state_time = pygame.time.get_ticks()
        self.state_duration = MOVE_FORWARD_TIME

    def check_transition(self, agent, state_machine):
        delta_time = get_delta_time(self.initial_state_time)
        if delta_time > self.state_duration:
            on_state_change(agent)
            agent.behavior.change_state(MoveInSpiralState())
        if agent.bumper_state:
            on_state_change(agent)
            agent.behavior.change_state(GoBackState())

    def execute(self, agent):
        agent.linear_speed = FORWARD_SPEED
  
```

Código 1: Classe MoveForwardState.

```

class MoveInSpiralState(State):
    def __init__(self):
        super().__init__("MoveInSpiral")
        self.initial_state_time = pygame.time.get_ticks()
        self.state_duration = MOVE_IN_SPIRAL_TIME

    def check_transition(self, agent, state_machine):
        delta_time = get_delta_time(self.initial_state_time)
        if delta_time > self.state_duration:
  
```

```

        agent.behavior.change_state(MoveForwardState())
        on_state_change(agent)
    if agent.bumper_state:
        on_state_change(agent)
        agent.behavior.change_state(GoBackState())

def execute(self, agent):
    delta_time = get_delta_time(self.initial_state_time)
    radios = INITIAL_RADIUS_SPIRAL + SPIRAL_FACTOR * delta_time
    agent.linear_speed = FORWARD_SPEED
    agent.angular_speed = agent.linear_speed / radios

```

*Código 2: Classe MoveInSpiralState.*

```

class GoBackState(State):
    def __init__(self):
        super().__init__("GoBack")
        self.initial_state_time = pygame.time.get_ticks()
        self.state_duration = GO_BACK_TIME

    def check_transition(self, agent, state_machine):
        delta_time = get_delta_time(self.initial_state_time)
        if delta_time > self.state_duration:
            agent.behavior.change_state(RotateState())
            on_state_change(agent)

    def execute(self, agent):
        agent.linear_speed = BACKWARD_SPEED

```

*Código 3: Classe GoBackState.*

```

class RotateState(State):
    def __init__(self):
        super().__init__("Rotate")
        self.initial_state_time = pygame.time.get_ticks()
        self.state_duration = random.random() * TURN_AROUND_MAX_TIME

    def check_transition(self, agent, state_machine):
        delta_time = get_delta_time(self.initial_state_time)
        if delta_time > self.state_duration:
            agent.behavior.change_state(MoveForwardState())
            on_state_change(agent)

    def execute(self, agent):
        agent.angular_speed = ANGULAR_SPEED

```

*Código 4: Classe RotateState.*

```

def get_delta_time(initial_state_time):
    return (pygame.time.get_ticks() - initial_state_time) / 1000

```

*Código 5: Função get\_delta\_time.*

```

def on_state_change(agent):
    agent.bumper_state = False
    agent.angular_speed = 0
    agent.linear_speed = 0

```

*Código 6: Função on\_state\_change.*

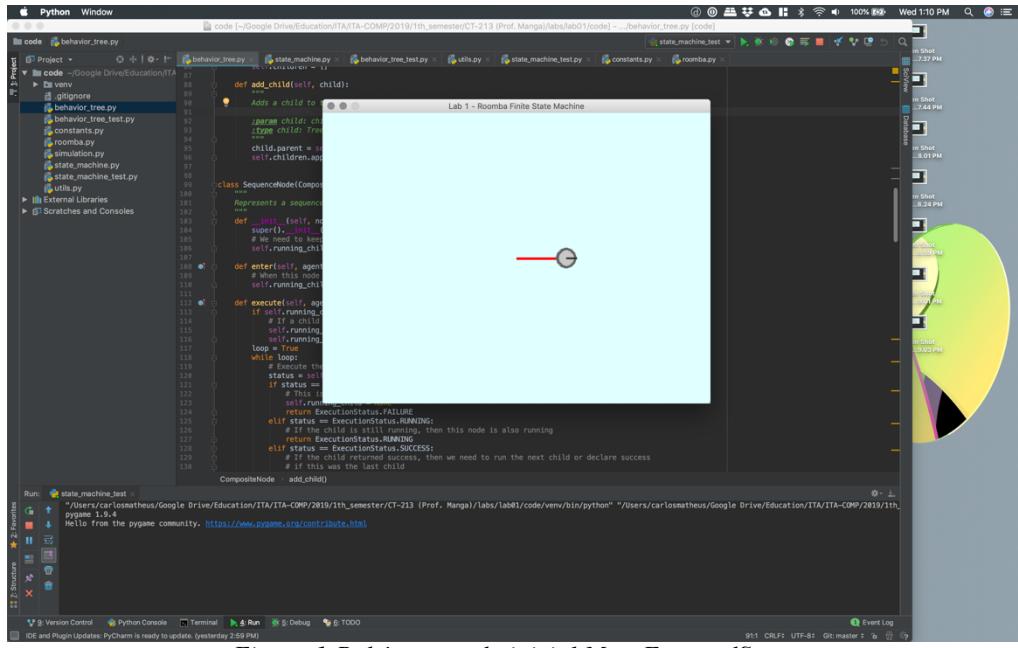


Figura 1:Robô no estado inicial MoveForwardState.

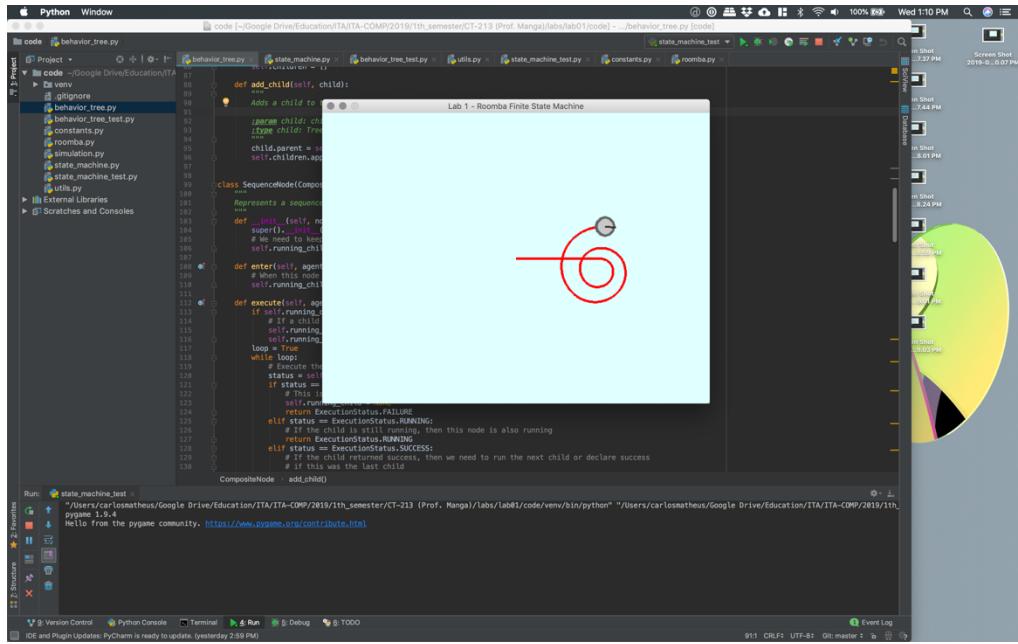


Figura 2: Robô no estado MoveInSpiralState.

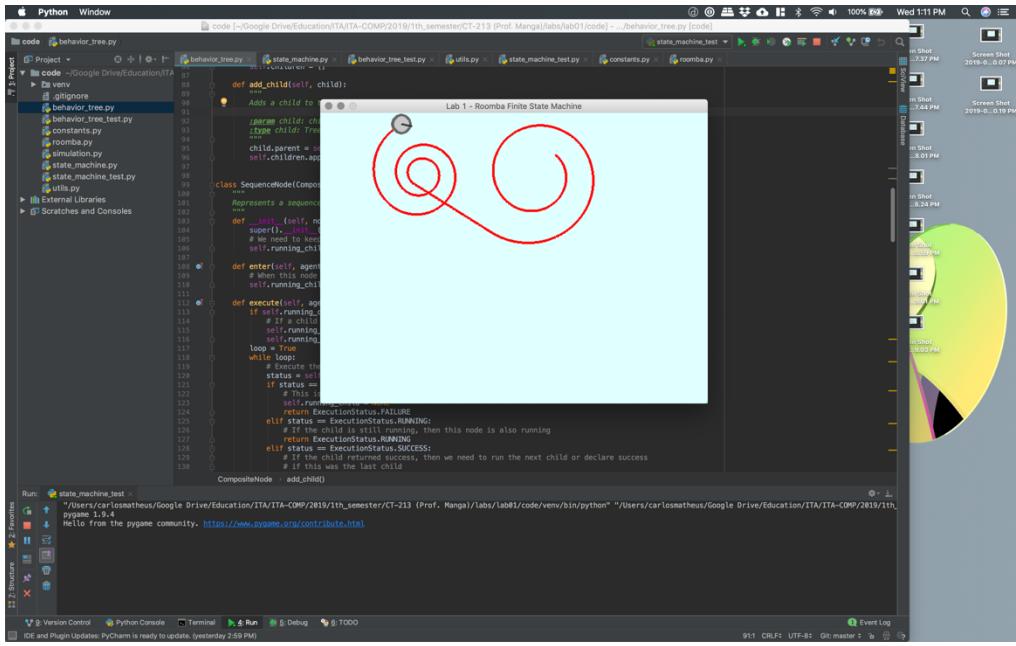


Figura 3: Robô após colidir e passar pelo estado GoBackState, agora está no estado RotateState

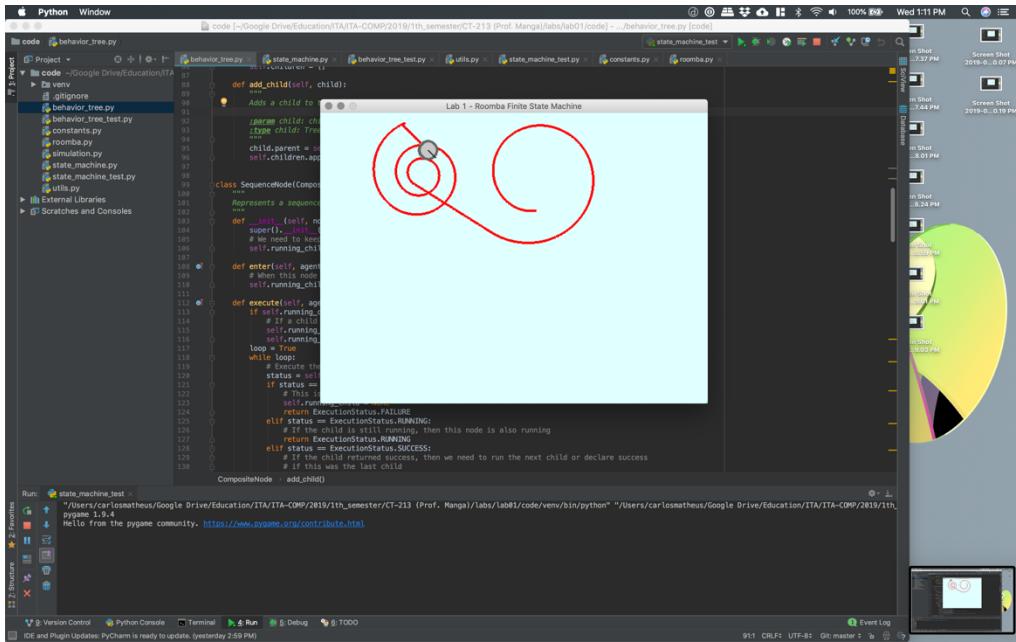


Figura 4: Robô após colidir e passar pelo estado RotateState retorna ao estado inicial MoveForwardState

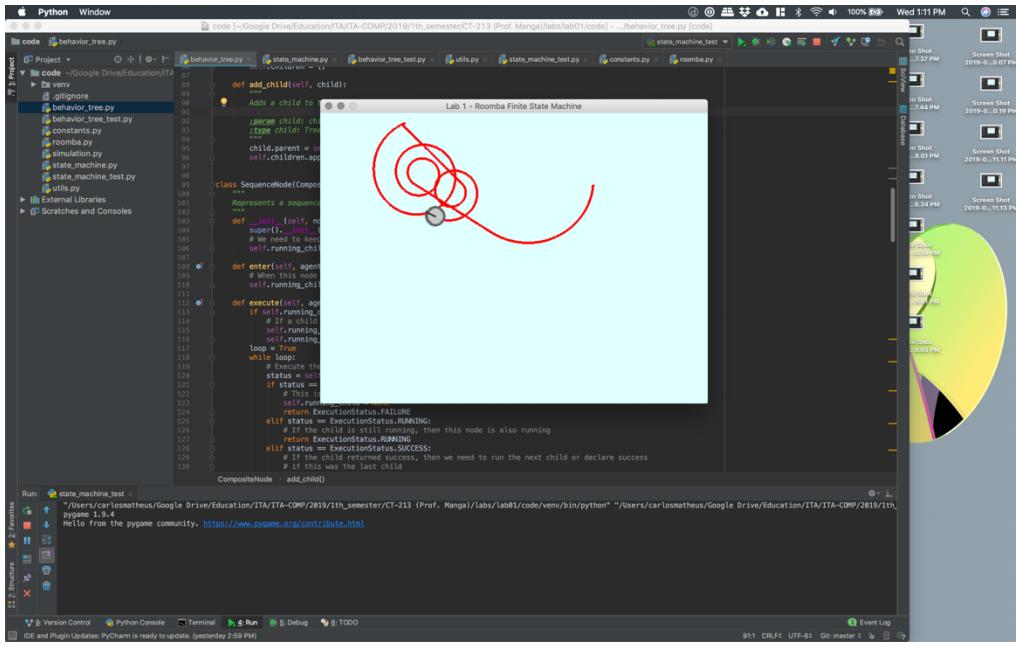
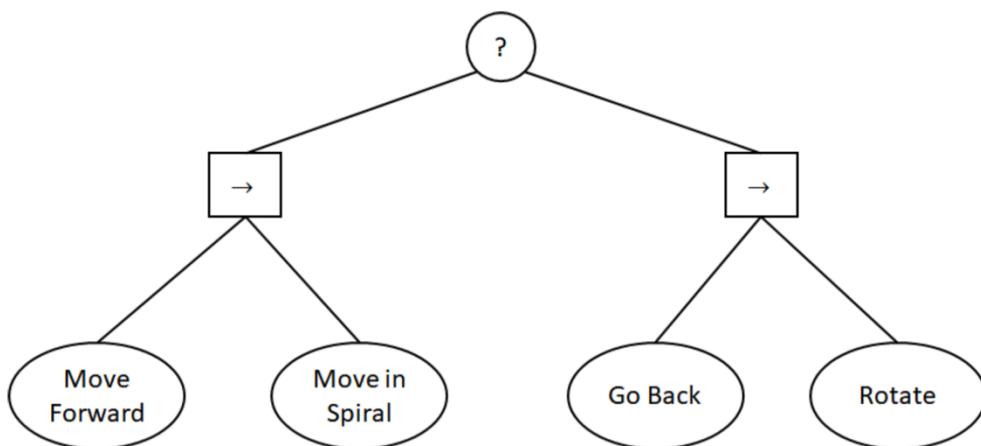


Figura 5: Robô após colidir e passar pelo estado MoveForwardState, vai para o estado RotateState

**Behavior Tree** – Primeiramente todos os nós da árvore, bem como suas conexões são definidas no construtor do RoombaBehaviorTree. Após isso foi definido cada uma das folhas da árvore. É possível reduzir o funcionamento de cada folha à função enter e a função execute. Na função enter são definidos os parâmetros iniciais para que se execute o funcionamento daquele estado e na função execute esses parâmetros podem ser atualizados caso seja necessária atualização nos parâmetros e são checadas condições para retorno do nó, que, então, retorna se ele ainda está em execução, se foi sucesso ou se foi execução falha. De modo que isso define o próximo nó que será executado.

A construção dos nós da árvore foram definidas de acordo com o Esquema 2.

A classe que define a árvore de comportamento do robô está implementa no Código 1 e a definição de cada uma das folhas da árvore está definida do Código 2 ao Código 5. Os códigos Código 6 e Código 7 se referem a funções auxiliares criadas e usada nas classes dos nós. As imagens mostradas da Figura 1 à Figura 8 denotam as ações do robô bem como suas transições de comportamento.



Esquema 2: Esquema da árvore de comportamento

```

class RoombaBehaviorTree(BehaviorTree):
    """
    Represents a behavior tree of a roomba cleaning robot.
    """
    def __init__(self):
        super().__init__()

        root_selection = SelectorNode('root_selection')
        idle_selector = SequenceNode('idle_selector')
        collision_selector = SequenceNode('collision_selector')

        root_selection.add_child(idle_selector)
        root_selection.add_child(collision_selector)

        idle_selector.add_child(MoveForwardNode())
        idle_selector.add_child(MoveInSpiralNode())

        collision_selector.add_child(GoBackNode())
        collision_selector.add_child(RotateNode())

        self.root = root_selection

```

*Código 1: Classe RoombaBehaviorTree. Definição da árvore no construtor.*

```

class MoveForwardNode(LeafNode):
    def __init__(self):
        super().__init__("MoveForward")
        self.initial_state_time = 0
        self.state_duration = 0

    def enter(self, agent):
        on_state_change(agent)
        self.initial_state_time = pygame.time.get_ticks()
        self.state_duration = MOVE_FORWARD_TIME
        agent.linear_speed = FORWARD_SPEED

    def execute(self, agent):
        delta_time = get_delta_time(self.initial_state_time)

        if delta_time > self.state_duration:
            return ExecutionStatus.SUCCESS
        elif agent.bumper_state:
            return ExecutionStatus.FAILURE
        else:
            return ExecutionStatus.RUNNING

```

*Código 2: Classe MoveForwardNode.*

```

class MoveInSpiralNode(LeafNode):
    def __init__(self):
        super().__init__("MoveInSpiral")
        self.initial_state_time = 0
        self.state_duration = 0

    def enter(self, agent):
        on_state_change(agent)
        self.initial_state_time = pygame.time.get_ticks()
        self.state_duration = MOVE_IN_SPIRAL_TIME
        agent.linear_speed = FORWARD_SPEED

```

```

def execute(self, agent):
    delta_time = get_delta_time(self.initial_state_time)
    radios = INITIAL_RADIUS_SPIRAL + SPIRAL_FACTOR * delta_time
    agent.angular_speed = agent.linear_speed / radios

    if delta_time > self.state_duration:
        return ExecutionStatus.SUCCESS
    elif agent.bumper_state:
        return ExecutionStatus.FAILURE
    else:
        return ExecutionStatus.RUNNING

```

*Código 3: Classe MoveInSpiralNode.*

```

class GoBackNode(LeafNode):
    def __init__(self):
        super().__init__("GoBack")
        self.initial_state_time = 0
        self.state_duration = 0

    def enter(self, agent):
        on_state_change(agent)
        self.initial_state_time = pygame.time.get_ticks()
        self.state_duration = GO_BACK_TIME
        agent.linear_speed = BACKWARD_SPEED

    def execute(self, agent):
        delta_time = get_delta_time(self.initial_state_time)

        if delta_time > self.state_duration:
            return ExecutionStatus.SUCCESS
        else:
            return ExecutionStatus.RUNNING

```

*Código 4: Classe GoBackNode.*

```

class RotateNode(LeafNode):
    def __init__(self):
        super().__init__("Rotate")
        self.initial_state_time = 0
        self.state_duration = 0

    def enter(self, agent):
        on_state_change(agent)
        self.initial_state_time = pygame.time.get_ticks()
        self.state_duration = random.random() * TURN_AROUND_MAX_TIME
        agent.angular_speed = ANGULAR_SPEED

    def execute(self, agent):
        delta_time = get_delta_time(self.initial_state_time)

        if delta_time > self.state_duration:
            return ExecutionStatus.SUCCESS
        else:
            return ExecutionStatus.RUNNING

```

*Código 5: Classe RotateNode.*

```

def get_delta_time(initial_state_time):
    return (pygame.time.get_ticks() - initial_state_time) / 1000

```

Código 6: Função `get_delta_time`.

```
def on_state_change(agent):
    agent.bumper_state = False
    agent.angular_speed = 0
    agent.linear_speed = 0
```

Código 7: Função `on_state_change`.

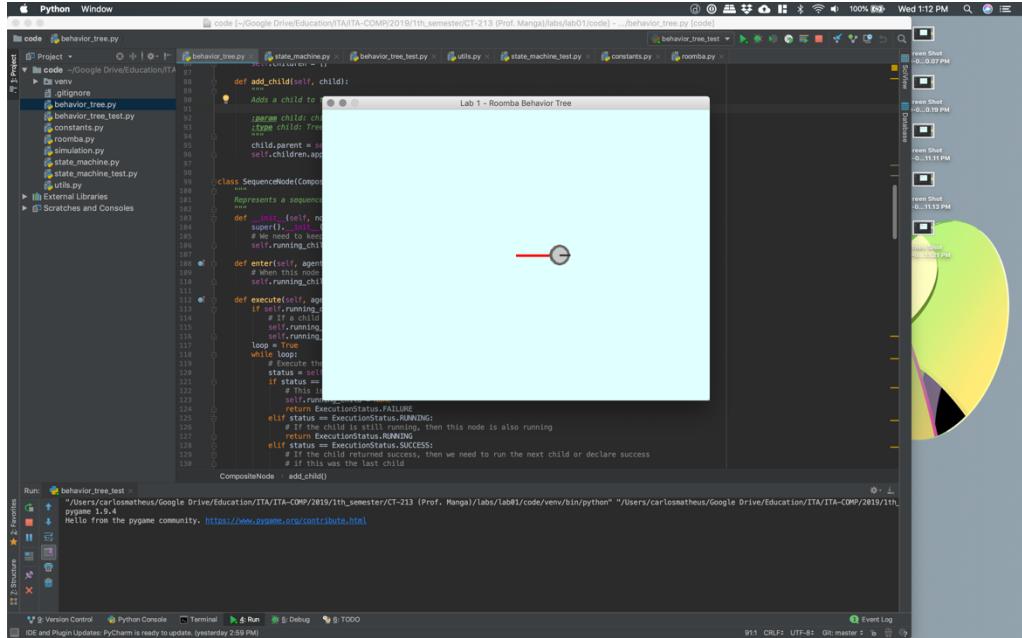


Figura 1: Robô no nó inicial `MoveForwardNode`.

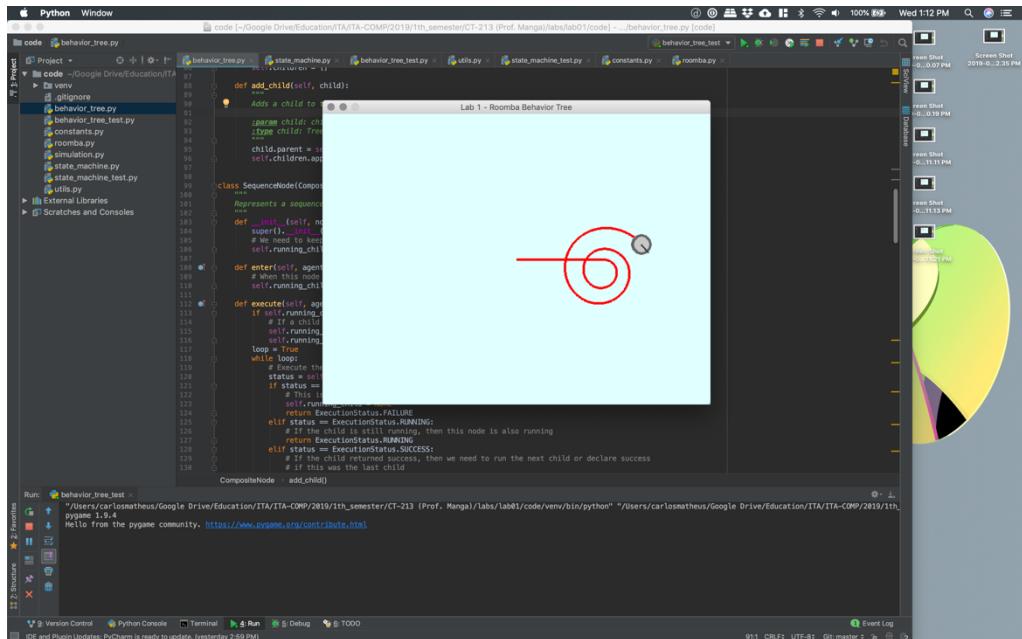


Figura 2: Robô no nó `MoveInSpiralNode`.

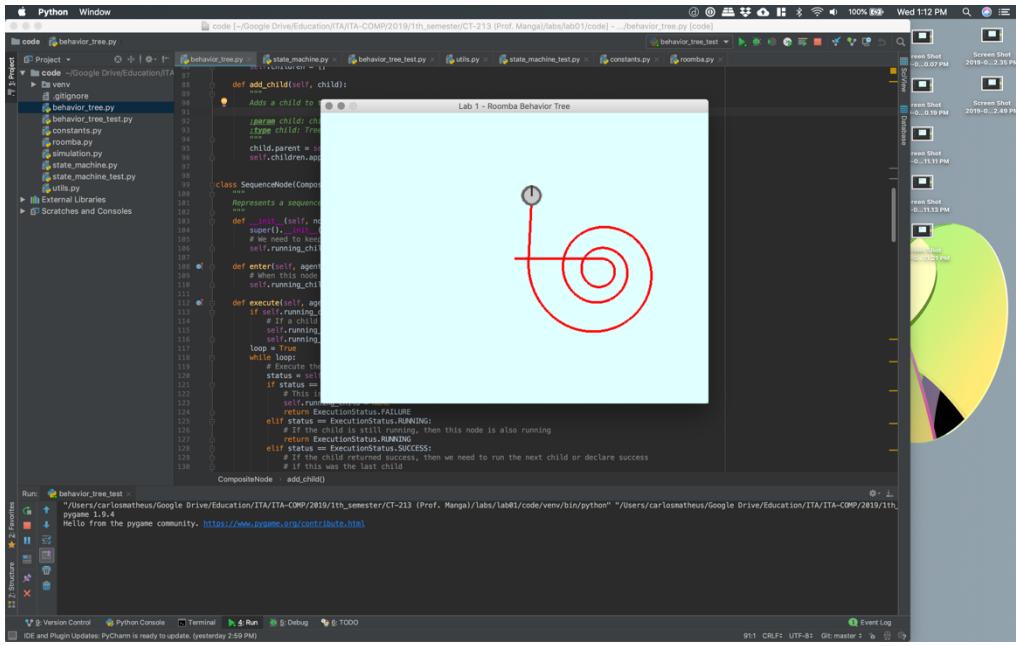


Figura 2: Robô retorna ao nó MoveForwardNode.

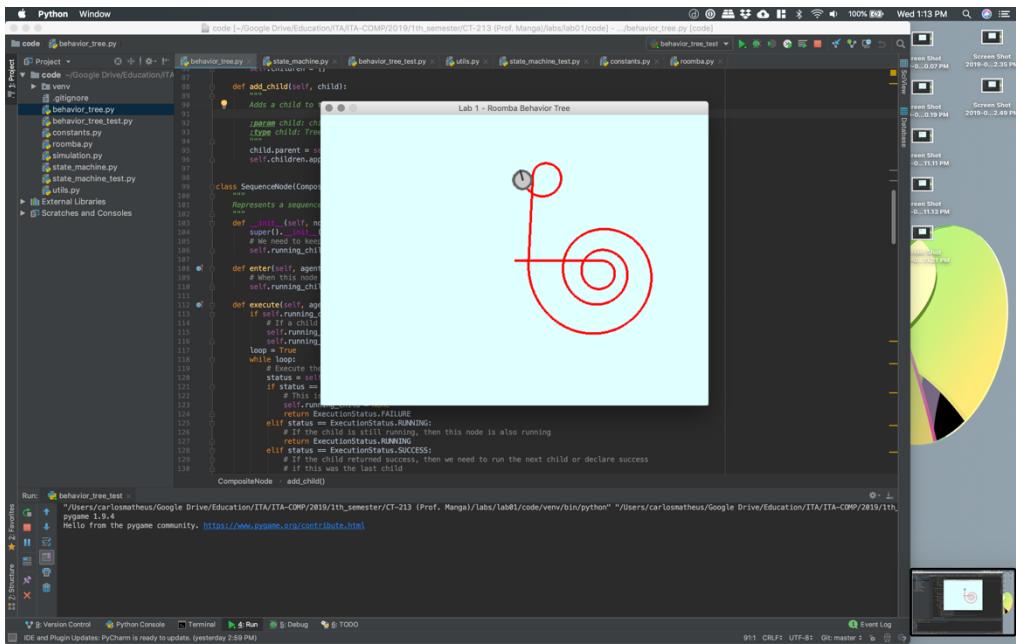


Figura 4: Robô retorna ao nó MoveInSpiralNode.

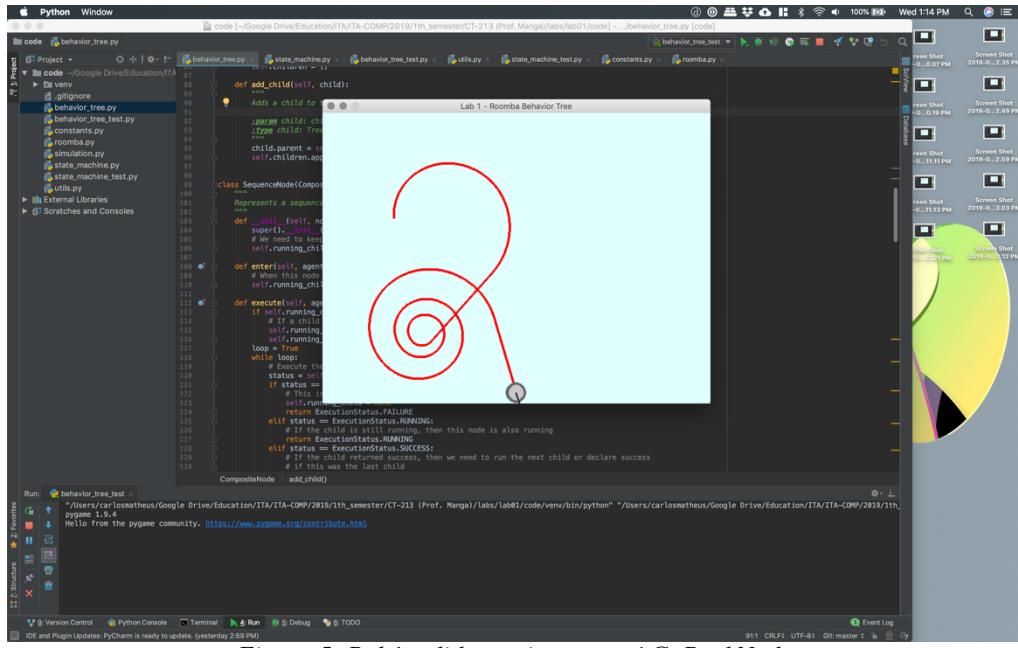


Figura 5: Robô colide e vai para o nó GoBackNode

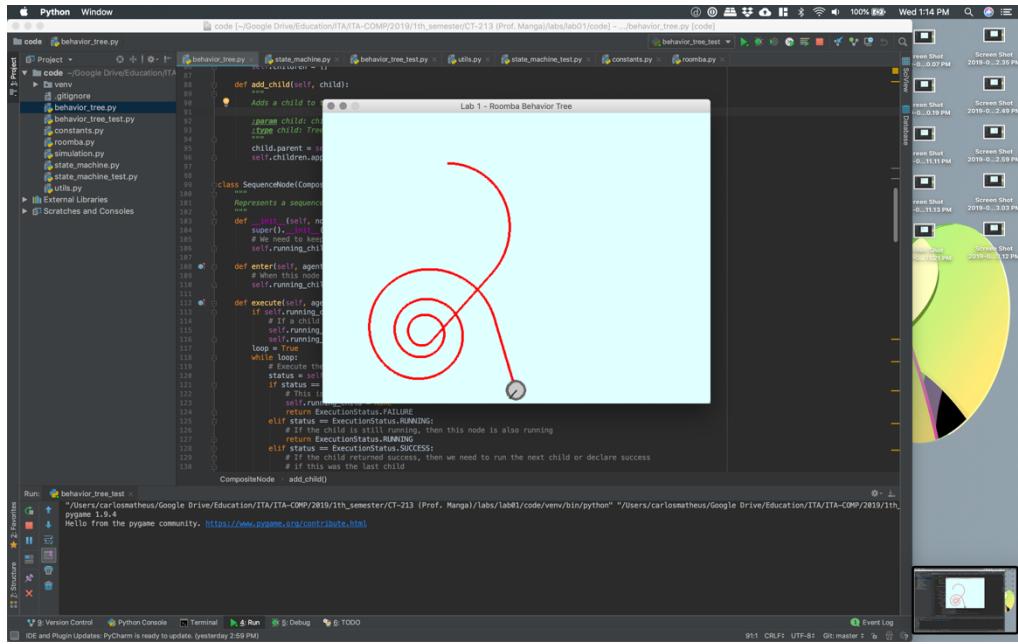


Figura 3:Robô após colidir e passar pelo nó GoBackNode, agora está no nó RotateNode.

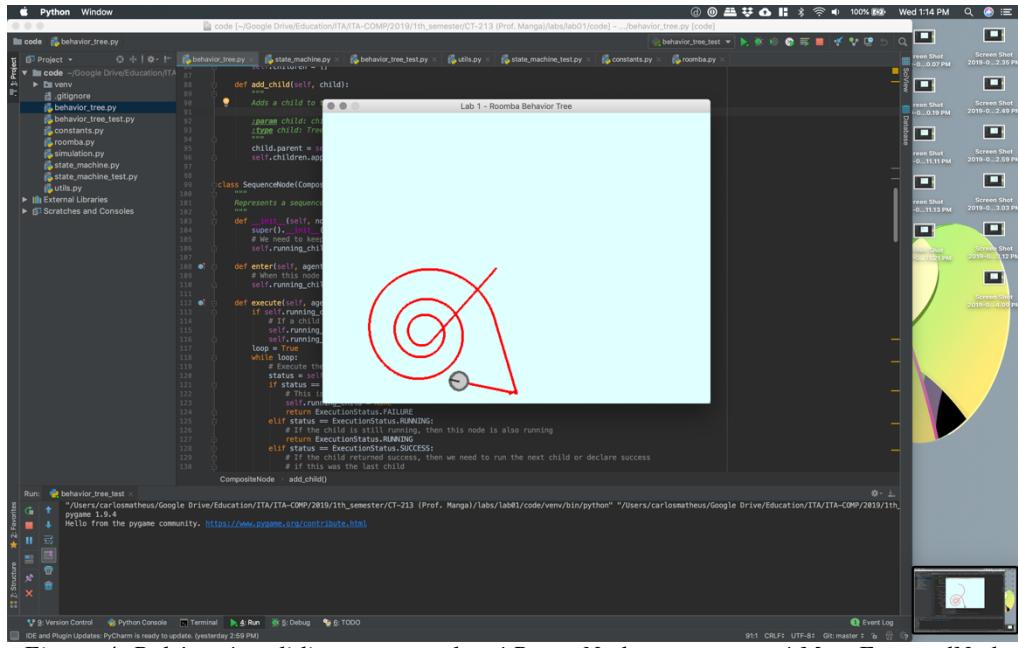


Figura 4: Robô após colidir e passar pelo nó RotateNode retorna ao nó MoveForwardNode

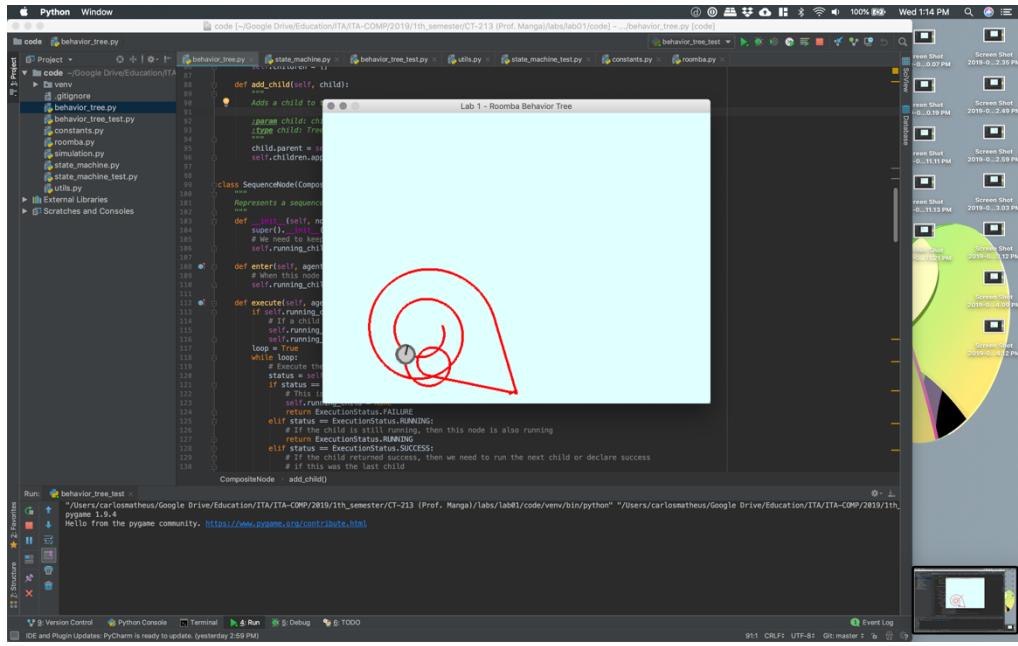


Figura 5: Robô após colidir e passar pelo nó MoveForwardNode, vai para o nó RotateNode.