Instituto Tecnológico de Aeronáutica – ITA

Inteligência Artificial para Robótica Móvel – CT-213

Professor: Marcos Ricardo Omena de Albuquerque Maximo

Aluno: Carlos Matheus Barros da Silva

# Lab 01:

O Lab foi desenvolvido com sucesso. Dividi a explicação relativa à maquina 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.

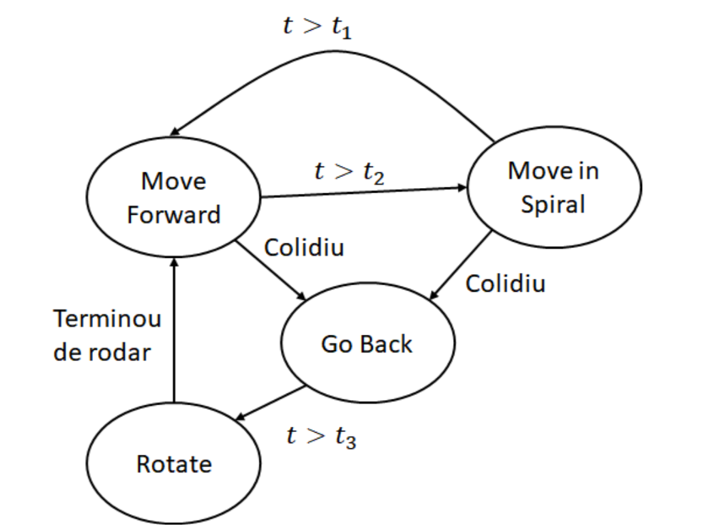
Digo isso principalmente porque estou usando MacOs e o pygame possui problema de otimização com o esse sistema operacional. Então existem momentos de queda de frame rate o que faz com que, devido a maneira que estava implementada a movimentação, o robô tivesse um comportamento de andar mais devagar ou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

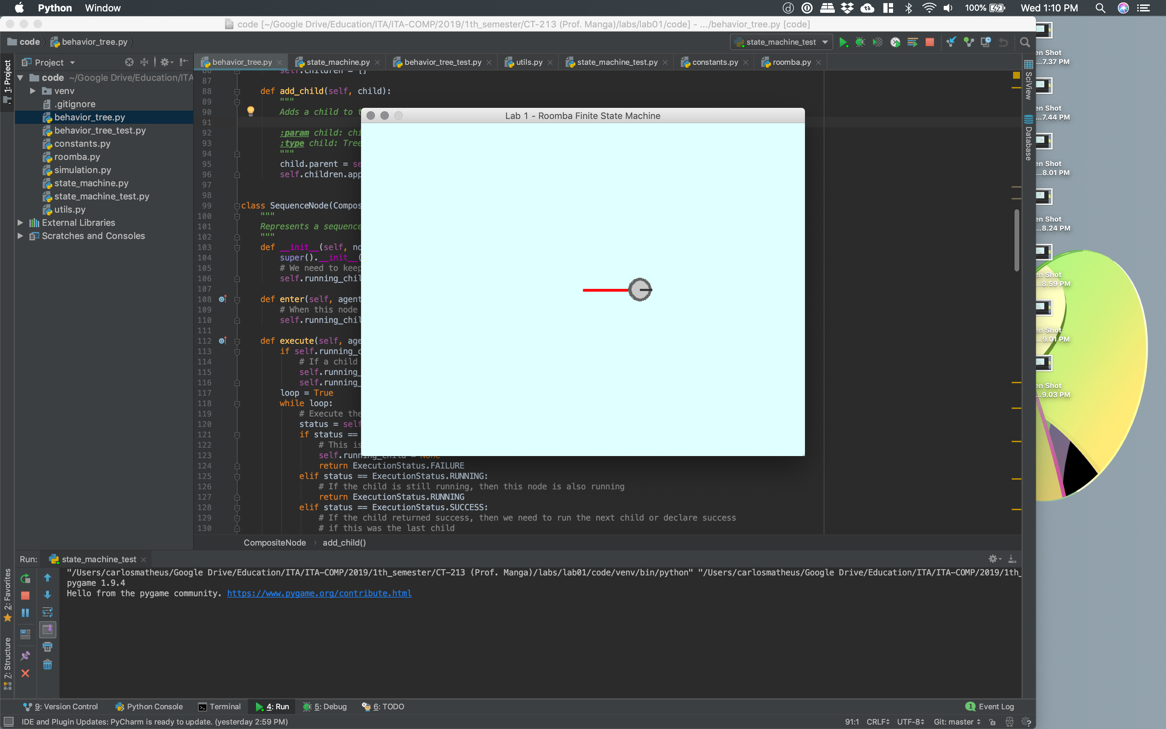
*Código 4: Classe RotateState.*

def get\_delta\_time(initial\_state\_time):  
 return (pygame.time.get\_ticks() - initial\_state\_time) / 1000

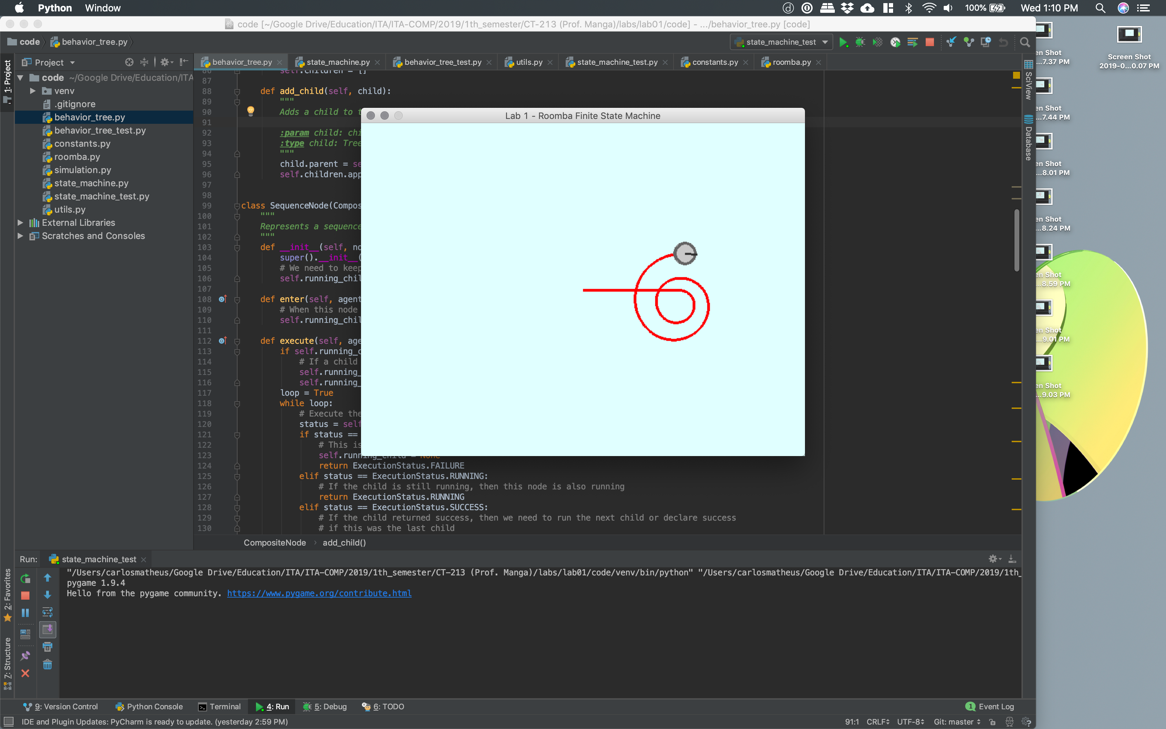
*Código 5: Função get\_delta\_time.*

def on\_state\_change(agent):  
 agent.bumper\_state = False  
 agent.angular\_speed = 0  
 agent.linear\_speed = 0

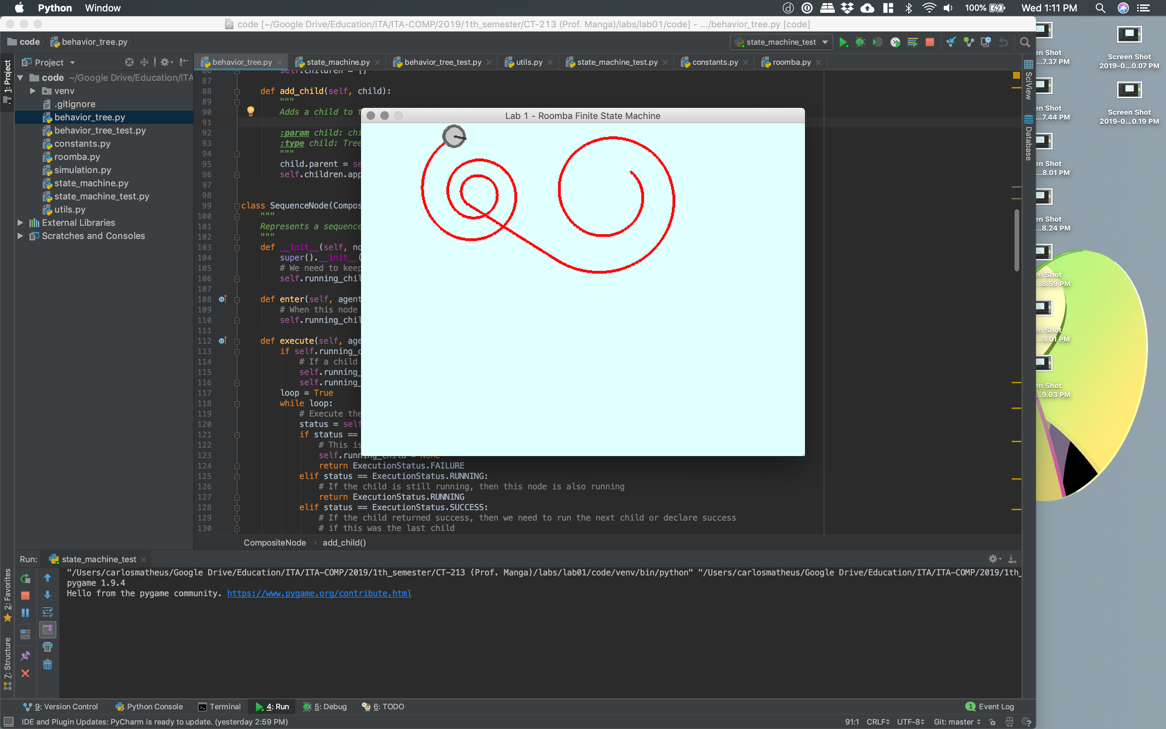
*Código 6: Função on\_state\_change.*



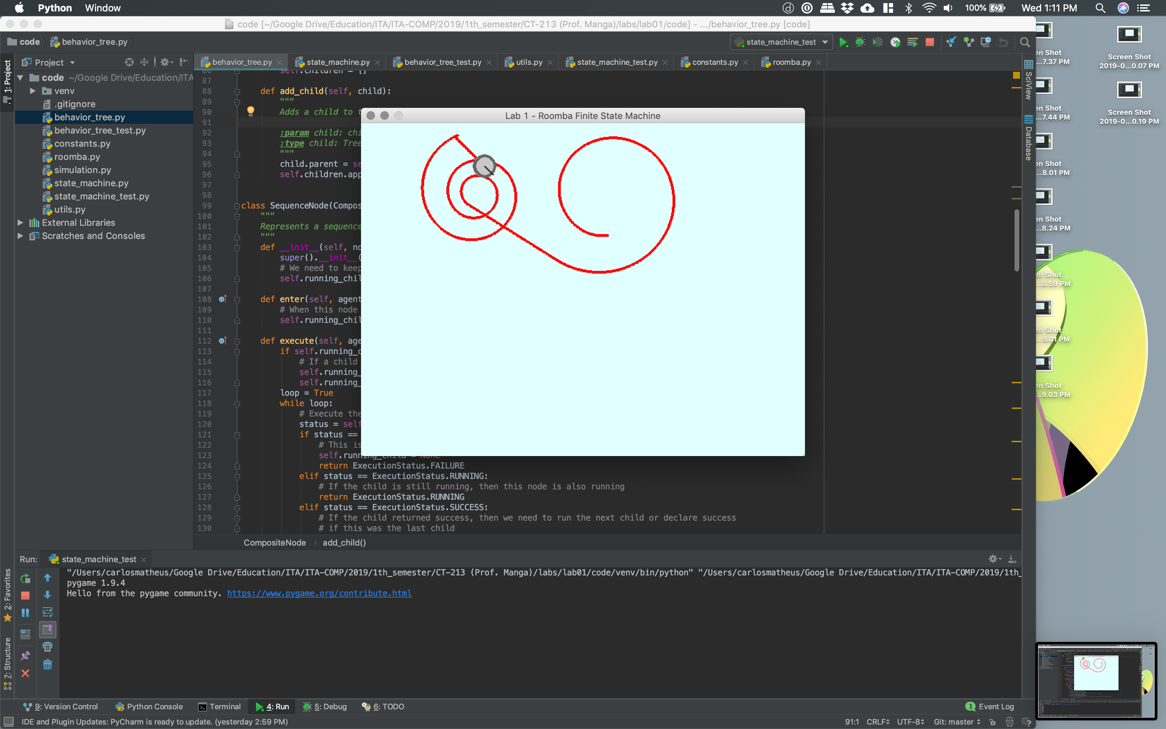
*Figura 1:Robô no estado inicial MoveForwardState.*



*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*

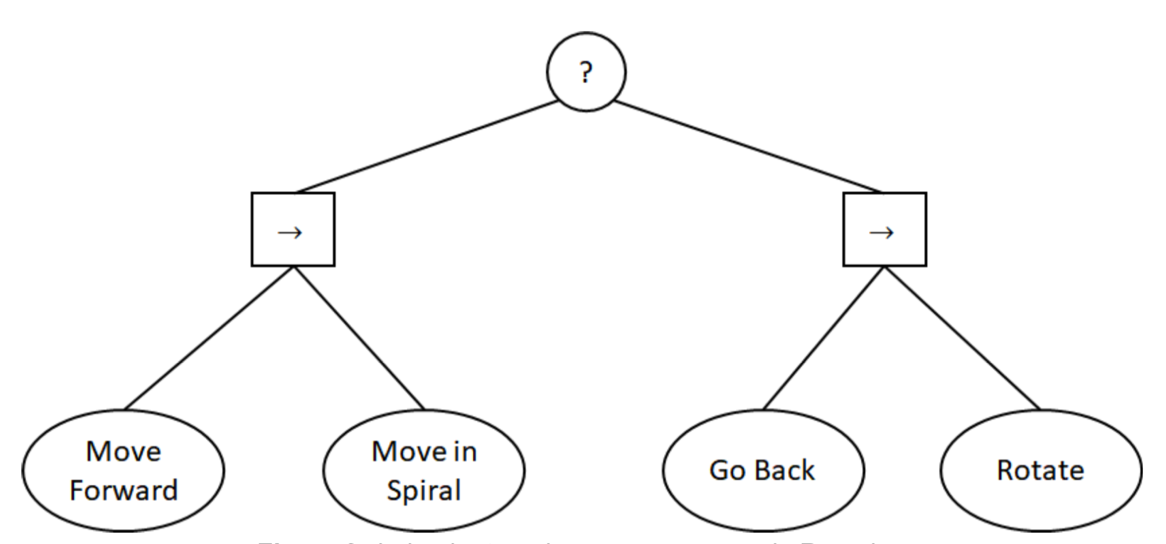


*Figura 5: Robô após colidir e passar pelo estado MoveForwardState, vai para o estado RotateState*

*Behavior Tree* – Primeiramente todos os nós da arvore, 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 parâmentros 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 arvore 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 arvore 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

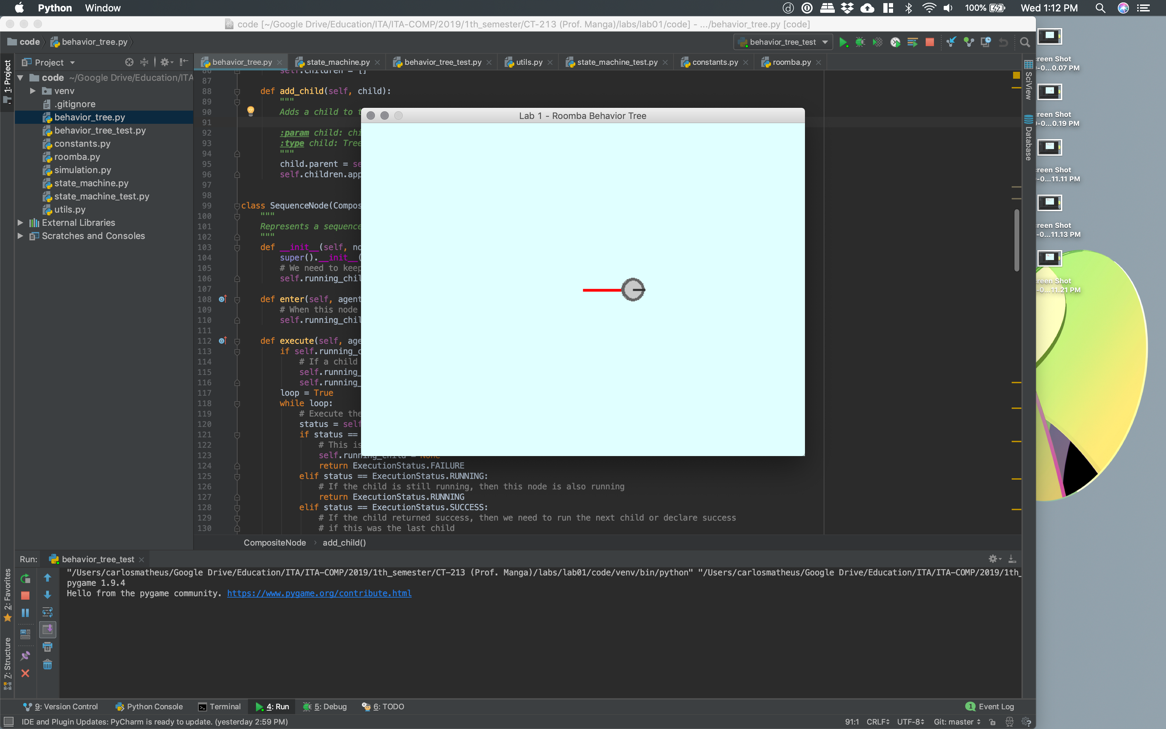
Código 5: Classe RotateNode.

def get\_delta\_time(initial\_state\_time):  
 return (pygame.time.get\_ticks() - initial\_state\_time) / 1000

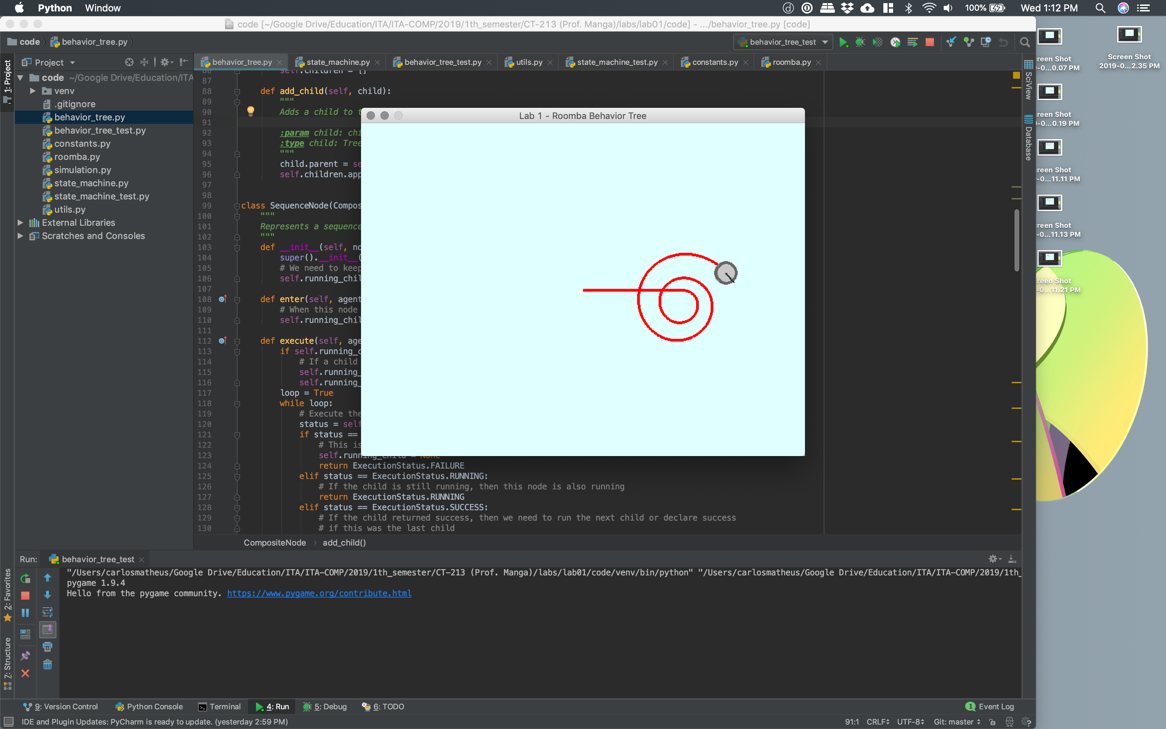
*Código 6: Função get\_delta\_time.*

def on\_state\_change(agent):  
 agent.bumper\_state = False  
 agent.angular\_speed = 0  
 agent.linear\_speed = 0

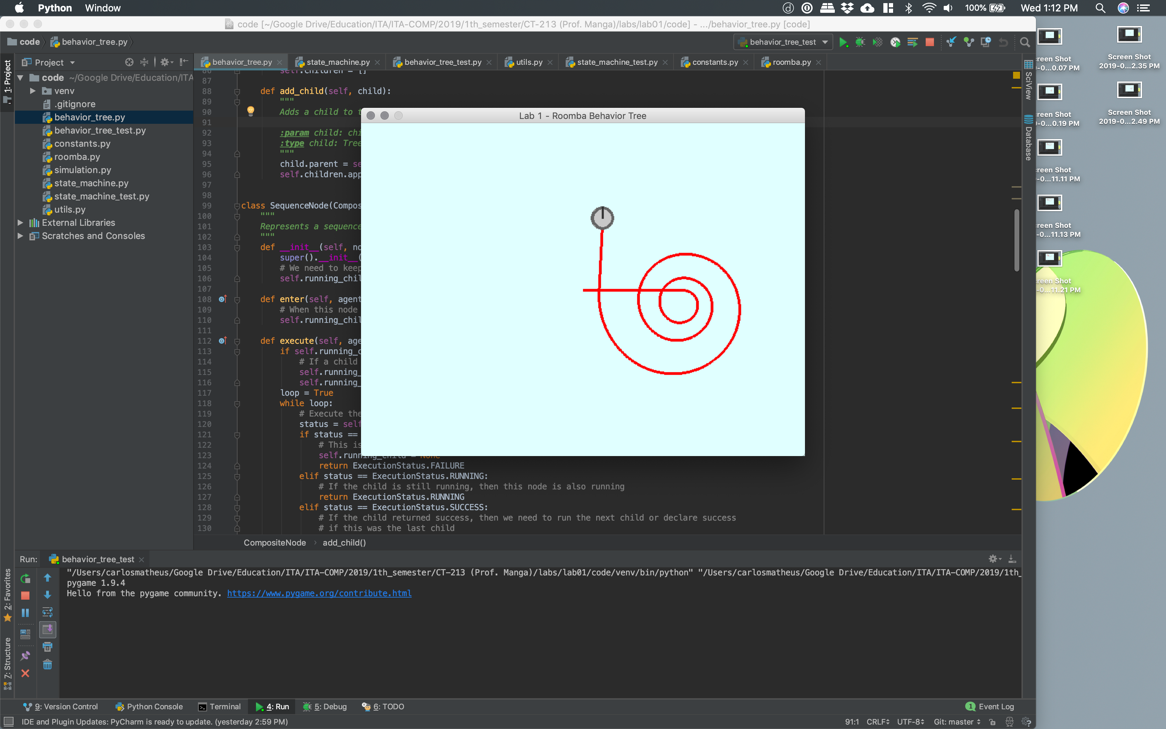
*Código 7: Função on\_state\_change.*



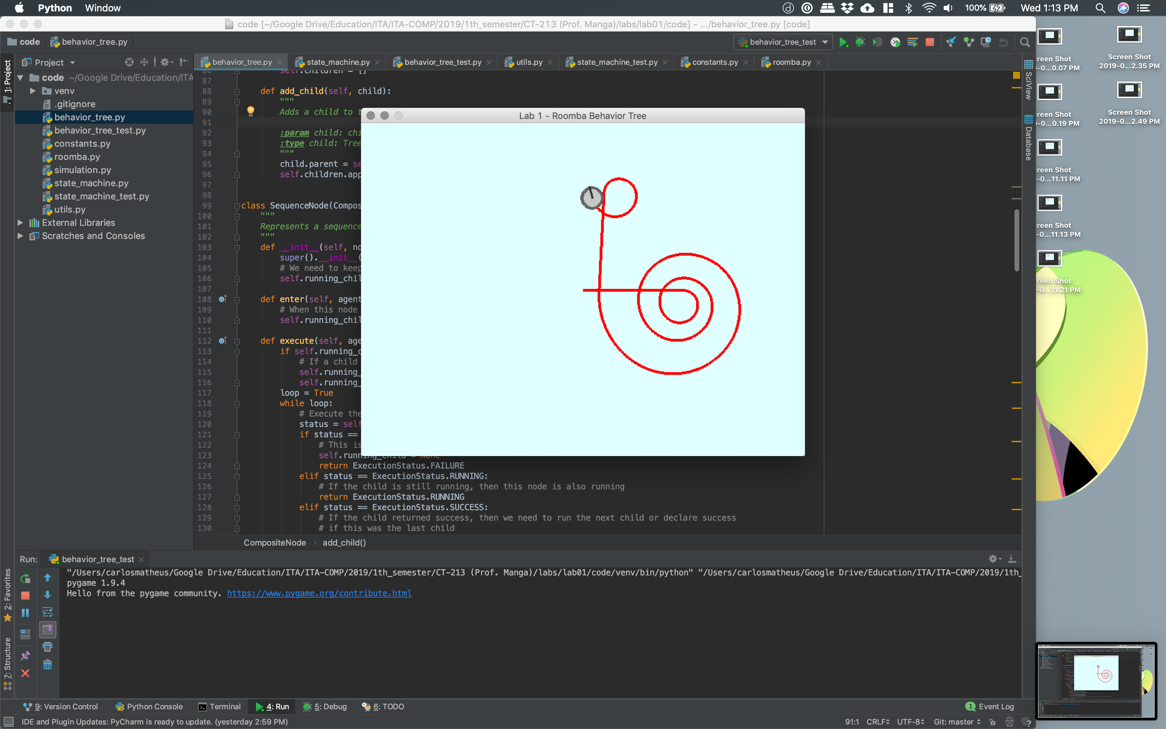
*Figura 1:Robô no nó inicial MoveForwardNode.*



*Figura 2: Robô no nó MoveInSpiralNode.*



*Figura 2: Robô retorna ao nó MoveForwardNode.*



*Figura 4: Robô retorna ao nó MoveInSpiralNode.*

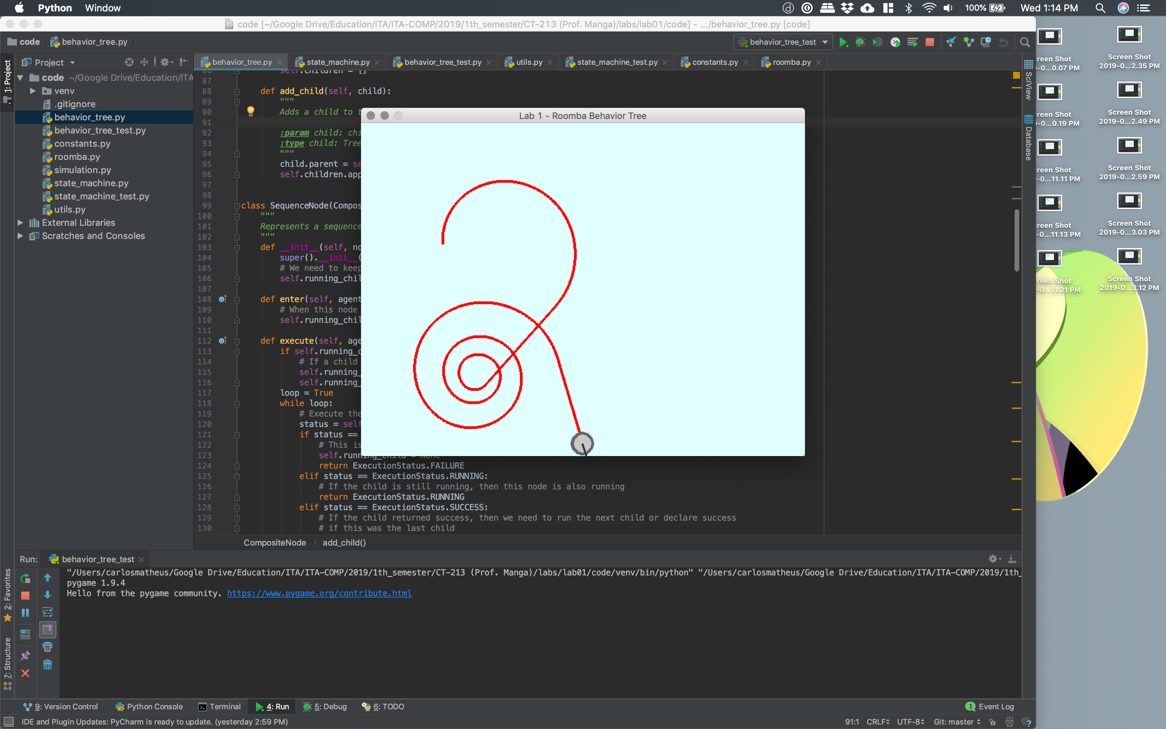
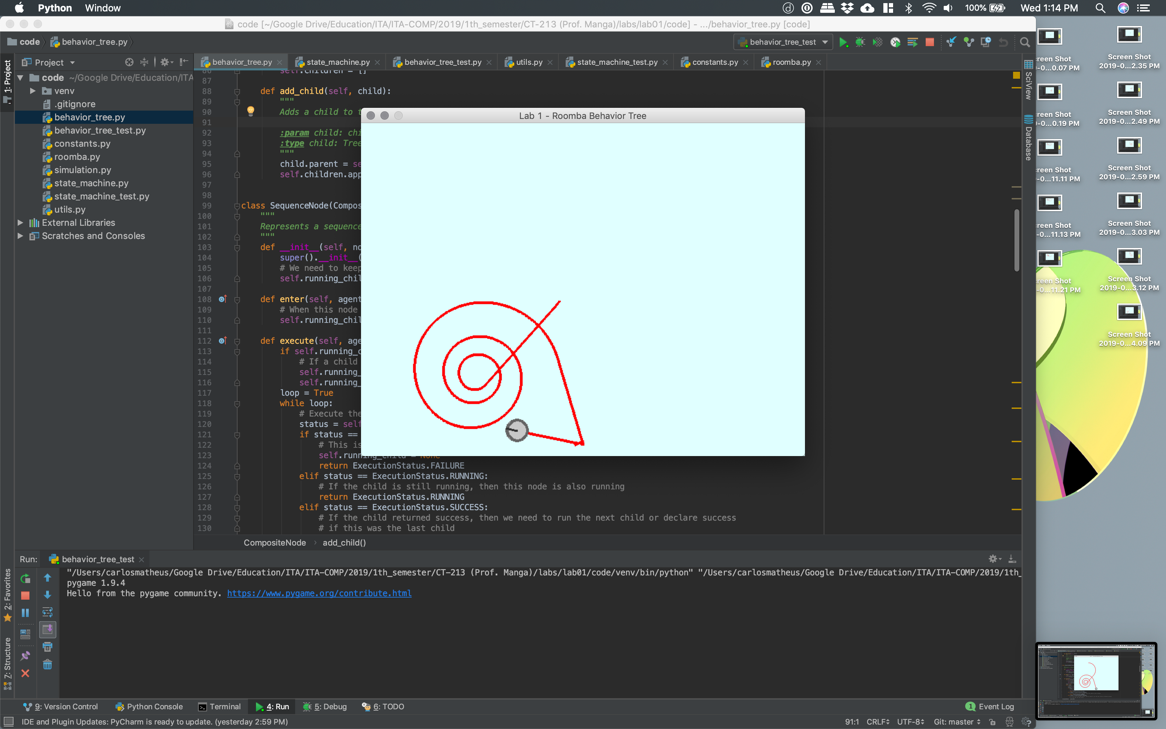


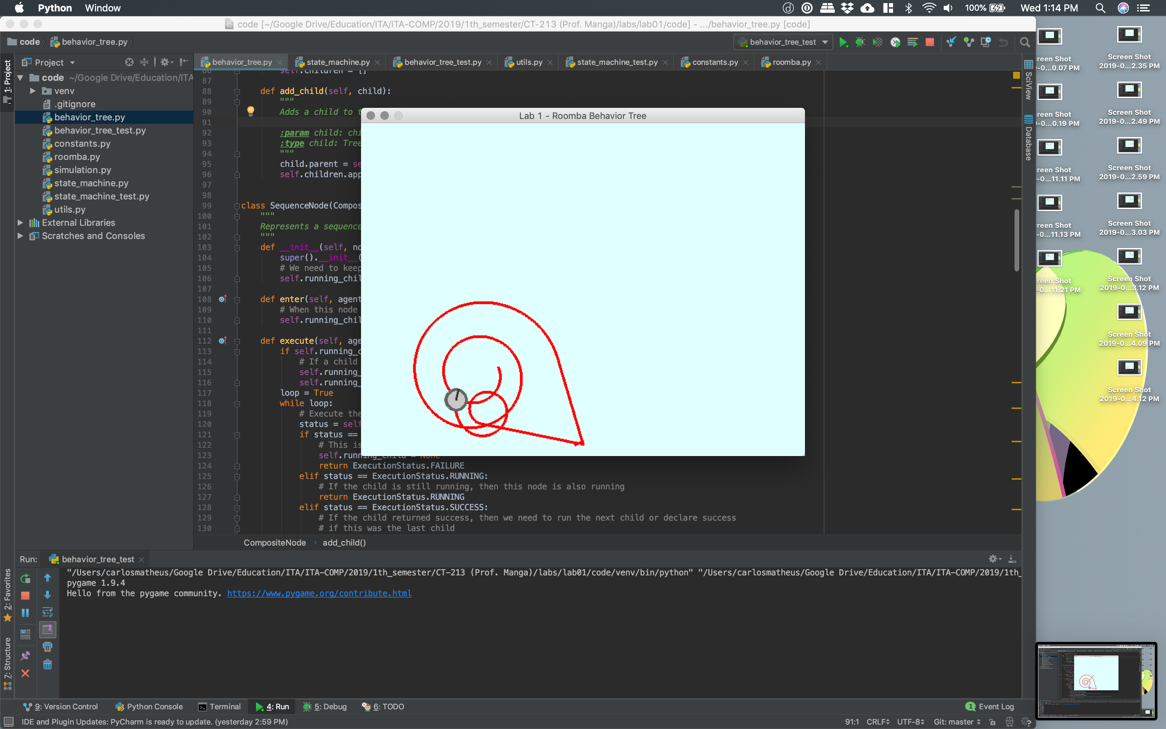
Figura 5: Robô colide e vai para o nó GoBackNode



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.*