Instituto Tecnológico de Aeronáutica
CT-213 Inteligência Artifical para Robótica Móvel
Marcos Ricardo Omena de Albuquerque Maximo

Laboratório 1: Máquina de Estados Finita e Behavior Tree

Débora Ferreira dos Santos

12 de Março de 2020

1 Implementação da máquina de estados

Nesta seção, são apresentados pontos relevantes da implementação da máquina de estados do Roomba. Primeiramente são discutidas as variáveis necessárias para realizar os movimentos descritos bem como avaliar a transição entre estados. Por fim, é apresentado um trecho de código para ilustrar as discussões.

1.1 Avaliando as tarefas

Para todas as movimentações, a execução da tarefa depende basicamente da redefinição das velocidades linear v e angular w. Esses valores permanecem constantes durante toda a execução do estado, exceto pelo estado *Move in spiral*. Para esse estado, embora se tenha uma velocidade angular inicial definida, é necessário atualizar o raio da espiral e, portanto, a velocidade angular a cada execução.

Portanto, como os valores são específicos para cada um dos estados, cada uma das classes é inicializada já com seu respectivo valor de velocidade. Para o movimento de espiral, os valores de velocidade são também atualizados no método *execute()*. Por fim, para a execução dos movimentos, no método *execute()*, são chamados o método do agente que define sua velocidade e o método que aciona seu movimento.

1.2 Avaliando as transições

As transições entre os estados seguem três categorias de eventos: colisão, tempo e conclusão da tarefa. A primeira tem implementação mais direta e é verificada quando o agente retorna valor *True* para o método *get_bumper_state()*.

Para as outras duas categorias, é preciso comparar o tempo total de execução e o tempo máximo de execução esperado. Para contabilizar o tempo total de execução de um determinado estado, considerou-se que ele é dado por

$$t = SAMPLE\_TIME * n, \tag{1}$$

em que $SAMPLE\_TIME$ é o tempo de amostragem da simulação e n é o número de execuções do estado.

Três das transições da máquina de estados são dadas por comparação desse valor t com uma constante definida para o problema. A implementação então é feita por uma estrutura condicional simples entre esses valores para verificar a necessidade de transição de estados.

O único caso ainda não coberto é o da conclusão da tarefa, que ocorre apenas após a rotação do robô por um ângulo aleatório. Nesse caso, para verificar o término da tarefa, foi definido o tempo máximo de execução associado ao movimento, calculado por

$$t_{max} = \frac{\theta_{rand}}{ANGULAR\_SPEED}, \tag{2}$$

em que θ_{rand} é um ângulo aleatório definido por um intervalo uniforme e $ANGULAR\_SPEED$ é uma constante do problema. Esse valor é definido na inicialização da classe *Rotate()*. Assim, para verificar a necessidade de transição basta comparar o tempo de execução com o valor de t_{max}.

Para contabilizar o tempo de execução de cada um dos estados, as classes são inicializadas com uma variável de tempo com valor igual a zero, representando o início da execução. O valor do tempo é incrementado a cada execução do método *execute()*, com base na Equação 1.

Por fim, essa variável de tempo também é utilizada para atualizar o raio do movimento de espiral e, em seguida, a velocidade angular conforme

$$\begin{aligned} r &= INITIAL\_RADIUS\_SPIRAL + SPIRAL\_FACTOR * t \\ w &= \frac{v}{r}. \end{aligned} \tag{3}$$

1.3 Base da implementação

Cada classe tem sua particularidade como descrito anteriormente, mas todas seguem a mesma base de implementação apresentada na Figura 1.3.

```
class CurrentState(State):
    def __init__(self):
        super().__init__("State")
        self.t = 0
        self.v = constants.LINEAR_SPEED
        self.w = constants.ANGULAR_SPEED

    def check_transition(self, agent, state_machine):
        if self.t > constants.TIME:
            state_machine.change_state(NextState())
        else:
            pass

    def execute(self, agent):
        agent.set_velocity(self.v, self.w)
        agent.move()
        self.t += constants.SAMPLE_TIME
```

Figura 1: Base da implementação de cada classe na máquina de estados

As principais diferenciações são os valores das variáveis da classe, as condições verificadas para transição e os cálculos para ajuste da movimentação.

2 Implementação da *behavior tree*

Para a *behavior tree*, toda a execução anteriormente associada a um estado passa a ser implementada agora nas folhas da árvore. A maior parte das considerações anteriormente feitas sobre esse problema para a máquina de estados permanece válida para a *behavior tree*. Há apenas algumas diferenças na forma de implementação.

Como as folhas são inicializadas apenas uma vez durante a execução do programa, o método *enter()* é utilizado na *behavior tree* para reinicializar as variáveis a cada nova chamada do nó. Portanto, todas as atribuições feitas no método *init()* passam a constar no método *enter()*.

Nessa implementação, não há transição direta entre as folhas, pois agora a transição é gerenciada pelos nós compostos. Assim, é necessário que o nó retorne ao seu pai um dos seguintes status: *SUCCESS* em caso de conclusão da tarefa, *FAILURE* em caso de colisão e *RUNNING* quando o agente ainda está executando sua tarefa. Associando esses status às categorias de transição mencionadas na Seção 1.2, tem-se o retorno de *SUCCESS* quando ocorre evento de tempo e de conclusão de tarefa. Já o retorno de *FAILURE* se dá quando há colisão. Finalmente, *RUNNING* se dá em caso contrário. Essas estruturas condicionais consideram os mesmo parâmetros da máquina de estados, porém são implementadas no método *execute()* logo após a movimentação do agente.

Por fim, a árvore é criada inicializando as folhas correspondentes a cada uma das movimentações. Depois são criados dois nós a partir da classe *SequenceNode()* e os respectivos filhos são associados a esses nós. Por último, a partir de *SelectorNode()* é criado o nó raiz cujos filhos são os outros dois nós *Sequence*.

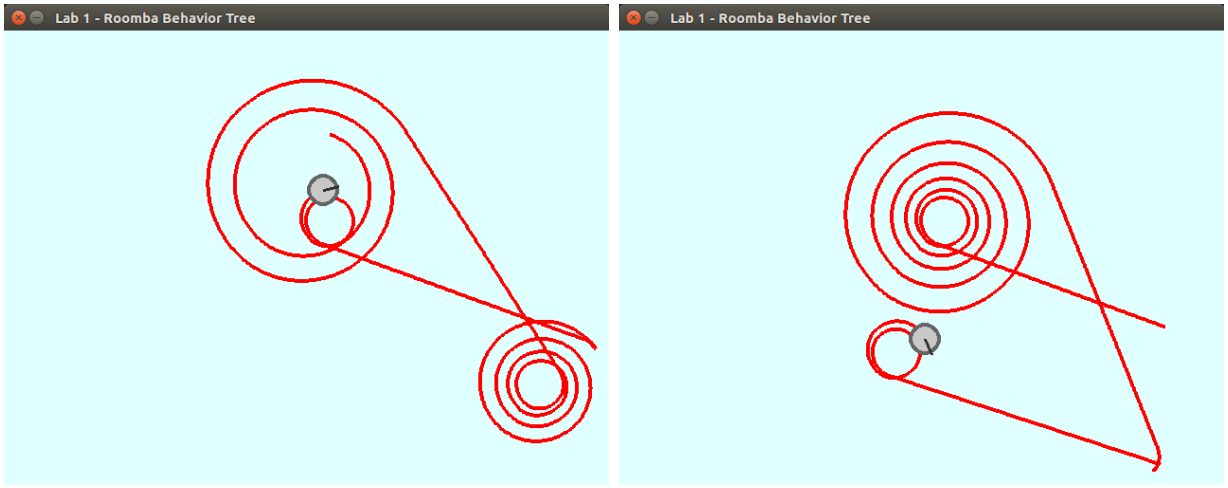
3 Resultados

A movimentação obtida por meio da implementação da máquina de estados pode ser vista na Figura 2. Já os resultados da implementação da *behavior tree* podem ser vistos na Figura 3.

Percebe-se que a movimentação corresponde à proposta inicial. O robô se movimenta em linha reta e, depois de um tempo, em espiral. Quando colide com as paredes, vai para trás e gira aleatoriamente até conseguir andar em linha reta novamente. Como esperado, visto que a lógica de ambas implementações é a mesma, os resultados para ambas as simulações são bastante similares.

(a) Instante 1 (b) Instante 2

Figura 2: Imagens ilustrativas da movimentação do Roomba na simulação com máquina de estados



(a) Instante 1 (b) Instante 2

Figura 3: Imagens ilustrativas da movimentação do Roomba na simulação com *behavior tree*