INSTITUTO TECNOLÓGICO DE AERONÁUTICA
CT-213

Laboratório 1: Máquina de Estados Finita e *Behavior Tree*

Professor:
Marcos Ricardo Omena de Albuquerque Máximo

Aluna:
Thayná Pires Baldão

São José dos Campos, 16 de março de 2021

1 Máquina de estados finita

Para implementar a máquina de estados finita capaz de tomar decisões pelo agente, implementou-se os métodos `init()`, `check_transition()` e `execute()` das classes que representam os estados `MoveForwardState`, `MoveInSpiralState`, `GoBackState` e `RotateState`, de modo a desenvolver a máquina de estados apresentada na Figura 2 do Roteiro do Laboratório 1.

Vale ressaltar que em todos estados o tempo atual é obtido multiplicando-se o `SAMPLE_TIME` da simulação pelo número de vezes que o estado foi executado, o que é medido pelo contador `number_executions` que é inicializado no método `init()` e atualizado a cada iteração do método `execute()` de cada estado.

1.1 `MoveForwardState`

A classe `MoveForwardState` representa o estado de o robô ir para frente e, de acordo com a Figura 2 do Roteiro do Laboratório 1, as únicas transições associadas a este estado ocorrem caso o robô colida com uma parede ou caso o robô passe tempo suficiente indo para frente. Por causa disso, no método `check_transition()` desta classe checa-se, primeiramente, se o agente colidiu com o parede e, em caso verdadeiro, muda-se o estado da máquina para o `GoBackState`. Do contrário, checa-se se o agente passou um intervalo de tempo igual a `MOVE_FORWARD_TIME` andando para frente e, em caso verdadeiro, muda-se o estado da máquina para o `MoveInSpiralState`. Ademais, o método `execute()` seta a velocidade do agente de modo que ele ande em linha reta, isto é, com velocidade linear igual a `FORWARD_SPEED` e velocidade angular nula.

1.2 `MoveInSpiralState`

A classe `MoveInSpiralState` representa o estado de o robô se mover em espiral e, de acordo com a Figura 2 do Roteiro do Laboratório 1, as únicas transições associadas a este estado ocorrem caso o robô colida com uma parede ou caso o robô passe tempo suficiente se movendo em espiral. Por causa disso, no método `check_transition()` desta classe checa-se, primeiramente, se o agente colidiu com o parede e, em caso verdadeiro, muda-se o estado da máquina para o `GoBackState`. Do contrário, checa-se se o agente passou um intervalo de tempo igual a `MOVE_IN_SPIRAL_TIME` se movendo em espiral e, em caso verdadeiro, muda-se o estado da máquina para o `MoveForwardState`.

Ademais, o método `execute()` seta a velocidade do agente de modo que ele se mova em espiral. Para fazer isso, primeiramente, computa-se o raio atual da espiral, dado pela equação: $r(t) = r_o + b \cdot t$. Posteriormente, utilizando a equação $\omega = \frac{v}{R}$ e considerando que a velocidade linear do agente se mantém constante enquanto ele percorre a espiral é possível obter a velocidade angular do agente. Após obter essa informação é possível setar as velocidades linear e angular do agente para que ele percorra a espiral.

1.3 `GoBackState`

A classe `GoBackState` representa o estado de o robô se mover para trás após colidir com a parede e, de acordo com a Figura 2 do Roteiro do Laboratório 1, a única transição associada a este estado ocorre caso o robô passe tempo suficiente se movendo para trás. Por causa disso, no método `check_transition()` desta classe checa-se se o agente passou um intervalo de tempo igual a `GO_BACK_TIME` se movendo para trás e, em caso verdadeiro, muda-se o estado da máquina para o `RotateState`. Ademais, o método `execute()` seta a velocidade do agente de modo que ele ande em linha reta para trás, isto é, com velocidade linear igual a `BACKWARD_SPEED` e velocidade angular nula.

1.4 `RotateState`

A classe `RotateState` representa o estado de o robô girar um $d\theta$ aleatório com o intuito de encontrar uma direção que o retire da parede e, de acordo com a Figura 2 do Roteiro do Laboratório 1, a única transição associada a este estado ocorre quando o robô termina de girar este $d\theta$. Para implementar a checagem desta transição foi preciso computar alguns valores no método `init()` desta classe. Primeiramente, computou-se o $d\theta$ aleatório entre $[-\pi, \pi)$ que o agente iria girar. Posteriormente, computou-se o tempo necessário para o agente girar $d\theta$, por meio da equação $t = \frac{d\theta}{\omega}$ e armazenou-se este resultado na variável `rotate_time`.

Todavia, como o $d\theta$ aleatório poderia ser positivo ou negativo, foi preciso fazer um tratamento para calcular `rotate_time` de forma correta, isto é, garantindo que o tempo computado fosse positivo. Feito isso, para checar se o robô tinha terminado de girar, bastava verificar se o agente havia passado um intervalo de tempo igual a `rotate_time` girando. Em caso verdadeiro, mudava-se o estado da máquina para o `MoveForwardState`.

Ademais, como desejava-se que o robô girasse em sentido horário caso $d\theta$ fosse positivo e em sentido anti-horário caso $d\theta$ fosse negativo, ao invés de se utilizar `ANGULAR_SPEED` diretamente no código, criou-se a variável `angular_speed` que armazena a magnitude e direção de ω. Esta variável é inicializada no método `init()` da classe. Por fim, o método `execute()` seta a velocidade do agente de modo que ele gire em torno de si próprio, isto é, com velocidade linear nula e velocidade angular igual a `angular_speed`.

1.5 Teste

Na Figura 1 é possível observar capturas de tela consecutivas da simulação do *Roomba* utilizando a máquina de estados implementada neste laboratório. Em 1a e 1b é possível observar que o agente está alternando entre se mover para frente e em espiral, conforme é esperado. Já em 1c percebe-se que o agente está se recuperando de uma batida na parede, tendo voltado um pouco para trás e começado a girar para escolher uma nova direção para se movimentar. Em 1d, nota-se que o agente conseguiu encontrar uma direção favorável para sair da parede, se moveu em linha reta naquela direção e conseguiu recuperar seu movimento. Nas capturas seguintes observa-se que o agente volta a alternar entre se mover para frente e em espiral até que, eventualmente, ele encontra a parede, volta para trás e gira em uma direção aleatória para se livrar da parede, conforme o esperado.

2 *Behavior tree*

Para implementar a *behavior tree* capaz de tomar decisões pelo agente, implementou-se os métodos `init()`, `enter()` e `execute()` das classes que representam os nós `MoveForwardNode`, `MoveInSpiralNode`, `GoBackNode` e `RotateNode`. Ademais, criou-se a `RoombaBehaviorTree` de modo a desenvolver a *behavior tree* apresentada na Figura 3 do Roteiro do Laboratório 1.

Vale ressaltar que, em todos os nós, o tempo atual é obtido multiplicando-se o `SAMPLE_TIME` da simulação pelo número de vezes que o nó foi executado, o que é medido pelo contador `number_executions` que é declarado no método `init()`, inicializado no método `enter()` e atualizado a cada iteração do método `execute()` de cada nó.

2.1 `MoveForwardNode`

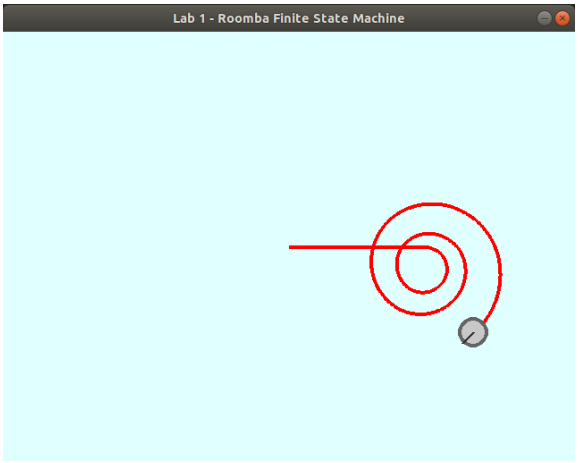
A classe `MoveForwardNode` representa o comportamento de o robô ir para frente. A cada execução deste comportamento checa-se, primeiramente, se o agente colidiu com o parede e, em caso verdadeiro, retorna-se o estado de execução `FAILURE`, dado que o agente não é mais capaz de se mover para frente. Esse retorno fará com que o pai desta subárvore (que é um *sequence*) retorne `FAILURE` para a raiz da *behavior tree* (que é um *selector*), a qual passará a executar comportamentos da subárvore da direita, responsável por recuperar o movimento do agente após este colidir com a parede.

Todavia, caso o agente não esteja colidindo com a parede, checa-se se o agente passou um intervalo de tempo igual a `MOVE_FORWARD_TIME` andando para frente e, em caso verdadeiro, retorna-se o estado de execução `SUCESS`, dado que o agente cumpriu sua tarefa de ir para frente. Esse retorno fará com que o agente mude para o comportamento `MoveInSpiralNode`.

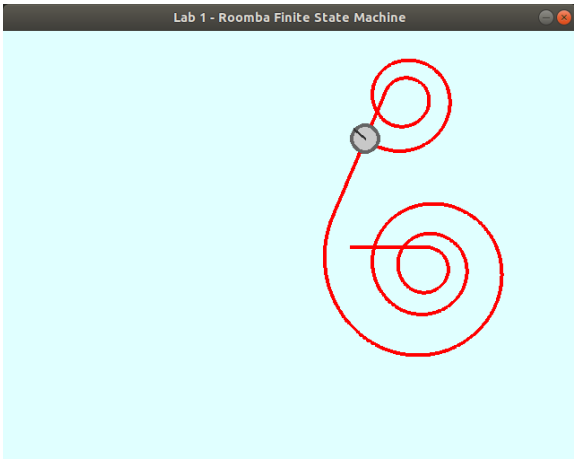
Por fim, caso nenhuma destas verificações seja verdadeira, apenas seta-se a velocidade do agente de modo que ele ande em linha reta e retorna-se o estado de execução `RUNNING` para que o agente continue executando aquela ação no próximo tempo de amostragem.

2.2 `MoveInSpiralNode`

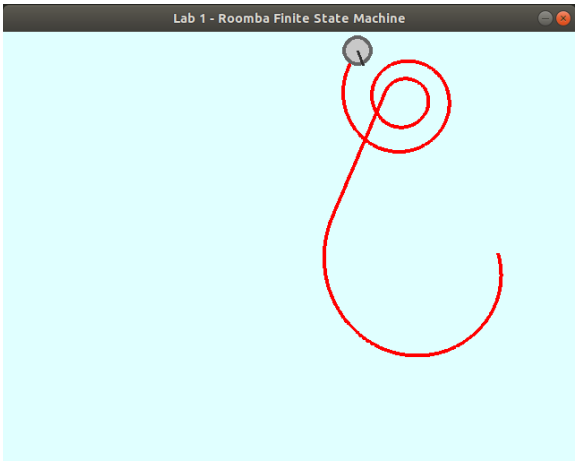
A classe `MoveInSpiralNode` representa o comportamento de o robô se mover em espiral. A cada execução deste comportamento, checa-se, primeiramente, se o agente colidiu com o parede e, em caso verdadeiro, retorna-se o estado de execução `FAILURE`, dado que o agente não é mais capaz de se mover em espiral. Esse



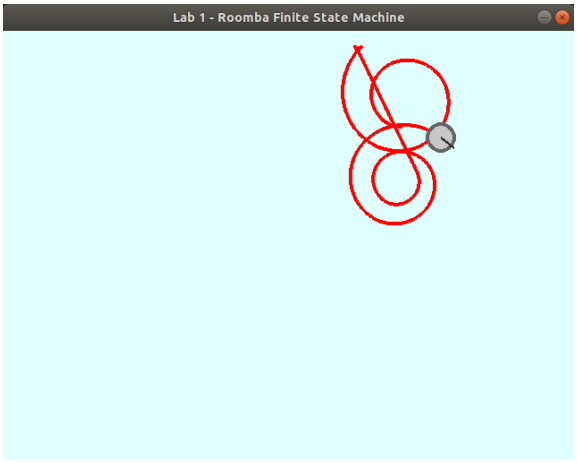
(a) Captura de Tela 1 - *Roomba FSM*



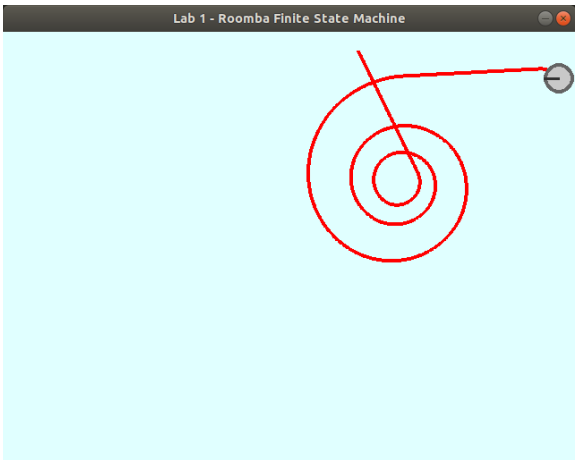
(b) Captura de Tela 2 - *Roomba FSM*



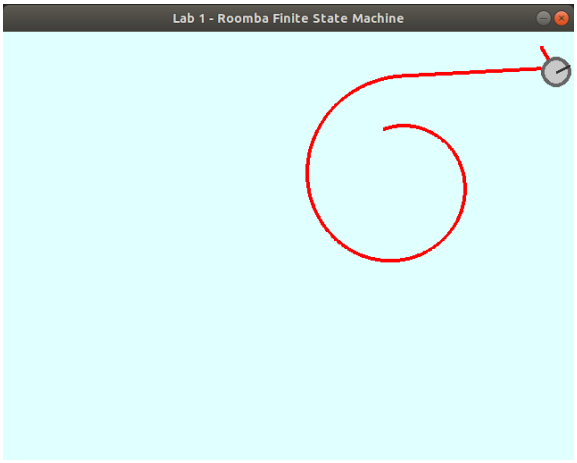
(c) Captura de Tela 3 - *Roomba FSM*



(d) Captura de Tela 4 - *Roomba FSM*



(e) Captura de Tela 5 - *Roomba FSM*



(f) Captura de Tela 6 - *Roomba FSM*

Figura 1: Capturas de tela da execução do *Roomba FSM*.

retorno fará com que o pai desta subárvore (que é um *sequence*) retorne `FAILURE` para a raiz da *behavior tree* (que é um *selector*), a qual passará a executar comportamentos da subárvore da direita, responsável por recuperar o movimento do agente após este colidir com a parede.

Todavia, caso o agente não esteja colidindo com a parede, checa-se se o agente passou um intervalo de tempo igual a `MOVE_IN_SPIRAL_TIME` se movendo em espiral e, em caso verdadeiro, retorna-se o estado de execução `SUCESS`, dado que o agente cumpriu sua tarefa de se mover em espiral. Esse retorno fará com que o pai de sua subárvore retorne `SUCESS` e, consequentemente, que a raiz da *behavior tree* retorne `SUCESS` para a raiz da *behavior tree*, o que fará com que a *behavior tree* reinicie sua execução pela subárvore esquerda. Isso garante que a alternação de comportamentos `MoveForwardNode` e `MoveInSpiralNode` continue.

Por fim, caso nenhuma destas verificações seja verdadeira, apenas seta-se a velocidade do agente de modo que ele se mova em espiral e retorna-se o estado de execução `RUNNING` para que o agente continue executando aquela ação no próximo tempo de amostragem. Para descobrir a velocidade angular necessária para executar a espiral é realizada a mesma sequência de passos utilizada na implementação da máquina de estados finita, descrita na Subseção 1.2.

2.3 `GoBackNode`

A classe `GoBackNode` representa o comportamento de o robô se mover para trás após colidir com a parede. A cada execução deste comportamento, checa-se se o agente passou um intervalo de tempo igual a `GO_BACK_TIME` se movendo para trás e, em caso verdadeiro, retorna-se o estado de execução `SUCESS`, dado que o agente cumpriu sua tarefa de se mover para trás. Esse retorno fará com que o agente mude para o comportamento `RotateNode`, dado que o pai deste nó é um *sequence*. Por fim, caso esta verificação não seja verdadeira, apenas seta-se a velocidade do agente de modo que ele ande em linha reta para trás e retorna-se o estado de execução `RUNNING` para que o agente continue executando aquela ação no próximo tempo de amostragem.

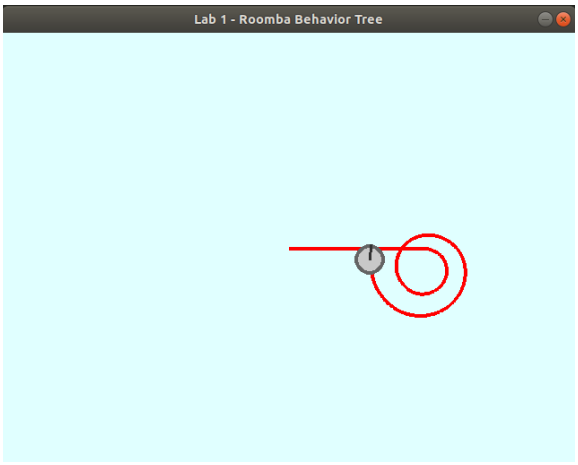
2.4 `RotateNode`

A classe `RotateNode` representa o comportamento de o robô girar um $d\theta$ aleatório com o intuito de encontrar uma direção que o retire da parede. A cada execução deste comportamento, checa-se se o agente passou um intervalo de tempo igual a `rotate_time` girando em torno de si próprio e, em caso verdadeiro, retorna-se o estado de execução `SUCESS`, dado que o agente cumpriu sua tarefa de girar um $d\theta$ aleatório. Esse retorno fará com que o pai de sua subárvore (que é um *sequence*) retorne `SUCESS` e, consequentemente, que a raiz da *behavior tree* (que é um *selector*) retorne `SUCESS` para a raiz da *behavior tree*, o que fará com que a *behavior tree* reinicie sua execução pela subárvore esquerda. Isso garantirá que o agente volte a alternar entre os comportamentos `MoveForwardNode` e `MoveInSpiralNode`.

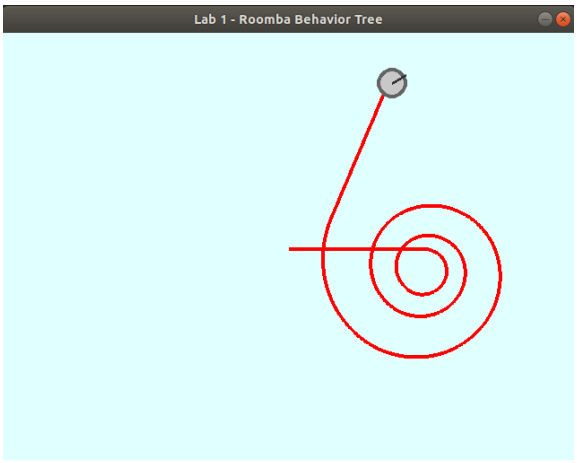
Por fim, caso nenhuma destas verificações seja verdadeira, apenas seta-se a velocidade do agente de modo que ele gire em torno de si próprio e retorna-se o estado de execução `RUNNING` para que o agente continue executando aquela ação no próximo tempo de amostragem. Para descobrir o `rotate_time` que o agente precisa girar para percorrer o $d\theta$ aleatório, assim como para se obter o valor correto da velocidade angular do robô, levando em consideração direção horária e anti-horária, é realizada, no método `enter()`, a mesma sequência de passos utilizada na implementação da máquina de estados finita, descrita na Subseção 1.4.

2.5 Teste

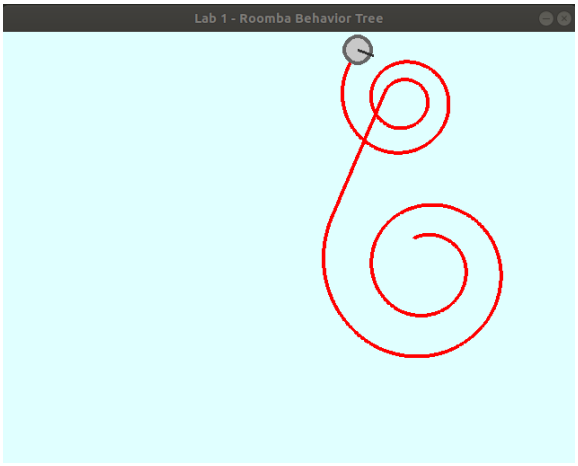
Na Figura 2 é possível observar capturas de tela consecutivas da simulação do *Roomba* utilizando a *behavior tree* implementada neste laboratório. Em 2a e 2b é possível observar que o agente está alternando entre se mover para frente e em espiral, conforme é esperado. Já em 2c percebe-se que o agente está se recuperando de uma batida na parede, tendo voltado um pouco para trás e tendo começado a girar para escolher uma nova direção para se movimentar. Em 2d, nota-se que o agente conseguiu encontrar uma direção favorável para sair da parede, se moveu em linha reta naquela direção e conseguiu recuperar seu movimento. Nas capturas seguintes observa-se que o agente volta a alternar entre se mover para frente e em espiral até que, eventualmente, ele encontra a parede, volta para trás e gira em uma direção aleatória para se livrar da parede, conforme o esperado.



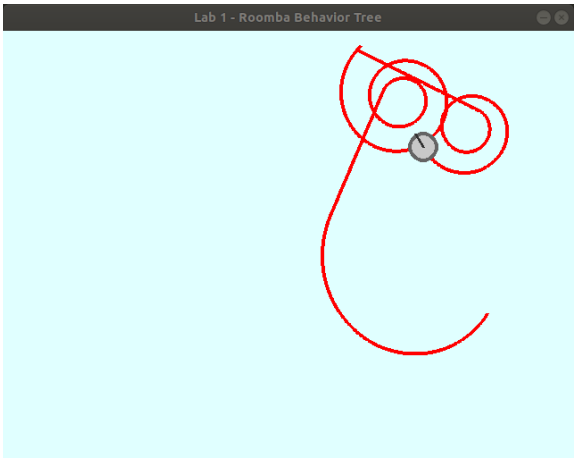
(a) Captura de Tela 1 - *Roomba BT*



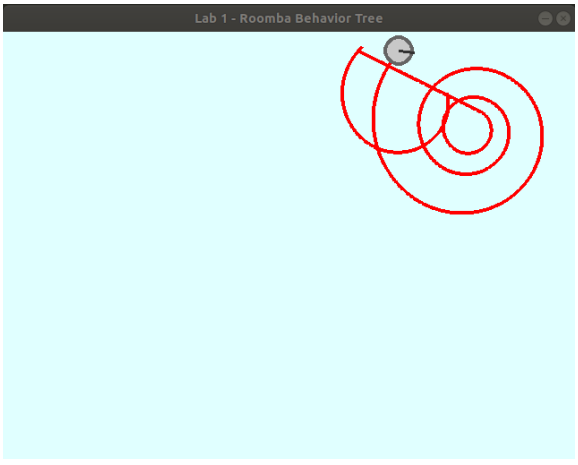
(b) Captura de Tela 2 - *Roomba BT*



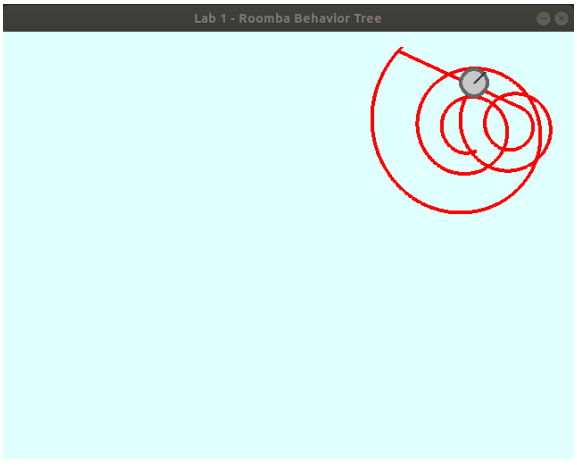
(c) Captura de Tela 3 - *Roomba BT*



(d) Captura de Tela 4 - *Roomba BT*



(e) Captura de Tela 5 - *Roomba BT*



(f) Captura de Tela 6 - *Roomba BT*

Figura 2: Capturas de tela da execução do *Roomba BT*.