U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

ARQUITETURA DE COMPUTAÇÃO EMBARCADA

PROJETO FINAL

ROBÔ SEGUIDOR DE LINHA

MESTRADO EM ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES

Duarte Ribeiro Afonso Branco up201905327@edu.fe.up.pt

Maria Inês Agostinho Simões up201904665@edu.fe.up.pt

INTRODUÇÃO

O presente relatório tem como objetivo documentar o processo de criação de um robô capaz de seguir uma trajetória plana demarcada por uma faixa escura. O trajeto poderá ter anomalias tais como a obstrução da faixa por um objeto ou uma falha, assim como bifurcações. Para além disso, deverá ser capaz de resolver um G*rid Maze*, com uma posição inicial e final definidas, navegando um percurso e reagindo à presença de obstáculos.

O robô é constituído por dois motores equipados com *enconders*, um *driver* capaz de suportar dois motores e um sistema de energia. Foram necessários, também, dois tipos de sensores na construção deste projeto, um sensor infravermelho capaz de distinguir a faixa escura do fundo claro e um sensor ultrassónico para evitar a colisão com obstáculos.

VÍDEO

Link para o vídeo do funcionamento do robô:

https://www.youtube.com/watch?v=pv-GP8dQEEI&ab_channel=DuarteBranco

HARDWARE

SENSORES INFRAVERMELHOS

De forma a possibilitar que o robô fosse capaz de distinguir entre a faixa escura e o plano de fundo claro que constituem o circuito, foram utilizados 5 sensores infravermelhos, denominados *5-ch ITR20001/T*, alimentados a *3.3V.*

A resposta destes, quando um objeto refletor (cor branca) é colocado à frente do sensor, corresponde a um pico de tensão pois o sinal infravermelho é refletido para o recetor. Se a cor do objeto for preta, a radiação infravermelha não é refletida e, portanto, a tensão é mínima. Assim, quando a linha preta é detetada no circuito, o sensor tem o valor de 0 e, em caso contrário, tem o valor de 1. Note-se para facilitar a implementação, estes valores foram invertidos, de modo a tomar o valor 1 quando a linha é detetada.

Com esta informação é possível verificar se o robô se encontra no local esperado, ou seja, em cima da faixa, assim como saber quando é necessário ajustar a posição do mesmo para que possa seguir o trajeto definido.

Para diminuir o número de condições que seriam necessárias criar para cobrir as várias permutações dos sensores, decidiu-se usar apenas o valor de uma média ponderada que pode ter valores dentro da gama de 0 até 5000, calculada da seguinte forma:

$$avg = \frac{in.IR1 * 1000 + in.IR2 * 2000 + in.IR3 * 3000 + in.IR4 * 4000 + in.IR5 * 5000}{in.IR1 + in.IR2 + in.IR3 + in.IR4 + in.IR5}$$

SENSORES ULTRASSÓNICO

Foi utlizado o sensor ultrassónico *Ultrasonic Ranging Module HC - SR04*, alimentado a *5V*. Aplicando um impulso de dez microssegundos no canal *TRIG* são enviados oito impulsos ultrassónicos pelo sensor a uma frequência de *40kHz*. Note-se que o sinal *ECHO* pode atingir os *5V* logo, tendo em conta que o *Raspberry Pi Pico* suporta um máximo de *3.3V*, foi necessário utilizar um divisor de tensão para permitir a leitura do sinal *ECHO*.

Consequentemente, é possível determinar a distância através da duração do sinal de entrada *ECHO*, utilizando a seguinte fórmula:

$$distance\_ultra = duration * 0.034 / 2$$

Através da duração do sinal do ECHO, que corresponde à ida e volta do sinal, e da velocidade do som ($340m/s = 0.034\ cm/s$) é possível determinar a distância ao objeto em centímetros.

Note-se que o sensor permite apenas detetar distâncias entre 2 centímetros e 4 metros e tem um ângulo de deteção de 15°.

MOTORES E DRIVER

Foram utlizados dois motores *DFRobot Micro DC*, alimentados entre *3V* e *7.5V*, e um dual motor *driver* do tipo *H-bridge* alimentado a *5V.* Através dos sinais *PWM* aplicados em AIN e BIN é possível definir o período de ativação dos interruptores e, consequentemente, a tensão e o sentido da corrente no motor, definindo-se o seu sentido de rotação, assim como a velocidade. Note-se que o *driver* utilizado não é o que consta na lista de material disponibilizada, mas sim um fornecido pelo professor Paulo Costa, pelo que tem um implementação ligeiramente diferente.

SISTEMA DE ENERGIA

Foram utlizadas duas pilhas *Li-Ion 18650* de *3,7V*, ligadas em série, logo com tensão de saída de *7.4V*. Assim sendo, tendo em conta que a alimentação dos vários dispositivos utilizados é de *5V*, foi necessário utilizar um conversor *Buck*, de modo a converter a tensão de *7.4V* em *5V*. Finalmente, para alimentar o *Raspberry Pi Pico*, foi necessário utilizar um díodo *Schottky*, de modo a assegurar que não ocorre o fenómeno de *back-powering* das pilhas quando o cabo *USB* é ligado em simultâneo.

ESQUEMA ELÉTRICO

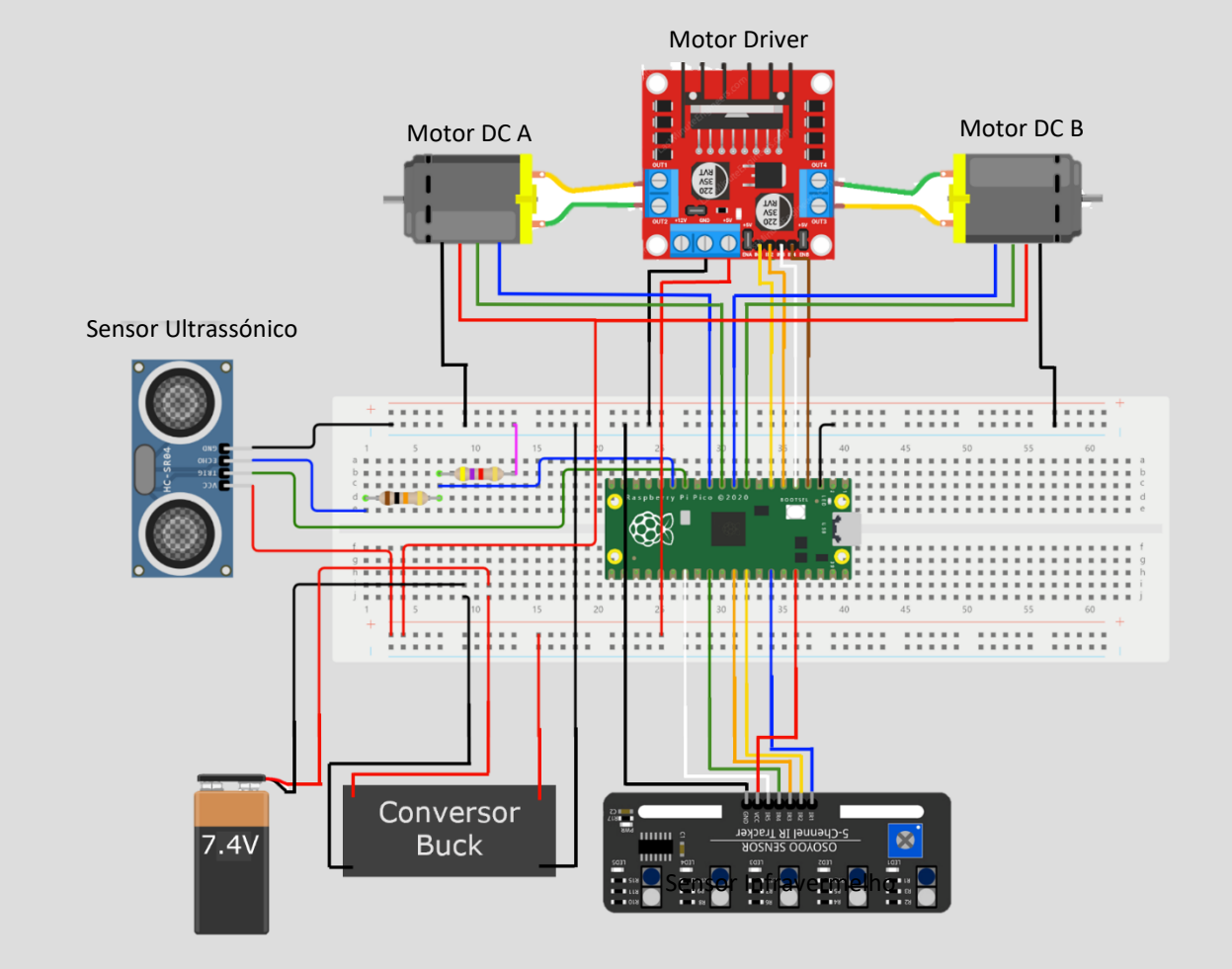

Figura 1 – Esquema Elétrico

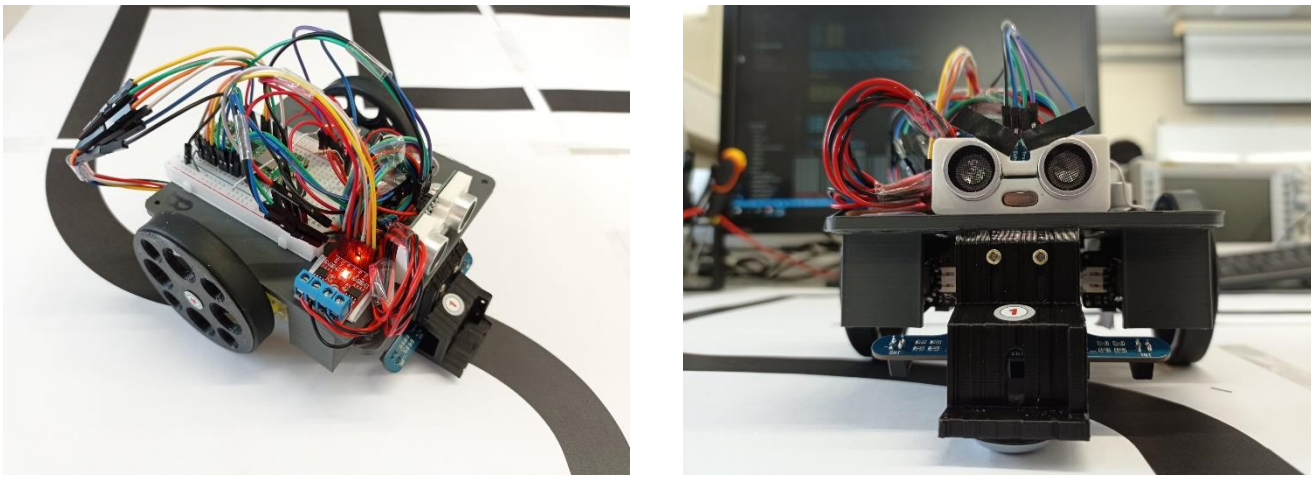

Figura 2 – Montagem do robô

SOFTWARE

Para o correto funcionamento do robô foram implementadas diferentes funções e máquinas de estado que permitem uma resposta apropriada aos diversos cenários traduzidos pelos sensores.

MOTORACTION()

Esta função foi implementada de modo a facilitar o controlo dos motores através do *driver*. Existem quatro modos de funcionamento F (Forward), B (Backwards), R (Right) e L (Left). No modo F o robô avança para a frente, enquanto no B avança para trás. Por sua vez, nos modos R e L o robô gira para a direita ou esquerda, respetivamente, com as rodas em sentidos opostos.

PID_SPEED()

Um controlador com ação proporcional, integral e derivativa (PID) é uma técnica de controlo que permite minimizar o erro, ajustar o *settling time*, o erro em regime permanente e o *overshoot*, respetivamente.

$K_p - ganho\ do\ controlador\ proporcional$

$K_i - ganho\ do\ controlador\ integral$

$K_d - ganho\ do\ controlador\ derivativo$

Os parâmetros foram sintonizados de forma adaptativa.

Estas constantes são depois multiplicadas por valores, tais como, P que é igual ao erro atual, o I é a soma de todos os erros e D corresponde à diferença entre o erro anterior e o atual. Note-se que o erro é nulo quando a média toma o valor de 3000, ou seja, o robô encontra-se corretamente alinhado com a faixa escura.

```
error_speed = 3000 - avg;
P_speed = error_speed;
I_speed += error_speed;
D_speed = error_speed - lastError_speed;
```

A equação final que permite determinar a diferença de velocidades:

```
speed = Kp_speed*P_speed + Ki_speed*I_speed + Kd_speed*D_speed;
speed1 = BASE_SPEED - speed;
speed2 = BASE_SPEED + speed;
```

Tem-se então que a velocidade denominada *speed1* corresponde à soma da velocidade base e da velocidade calculada e a *speed2* será a diferença entre a velocidade base e a velocidade calculada. É então possível realizar trajetórias curvilíneas com o modo F do *motorAction()*, uma vez que as rodas têm velocidades diferentes.

DIJKSTRA(), FINDPATH(), FILLADJ() E REMOVEADJ()

Estas funções são utlizadas para a resolução do *Grid Maze*, através do algoritmo de *Dijkstra*. Este algoritmo funciona percorrendo todos os nós de um grafo e associando a menor distância entre o nó atual e o inicial e o nó anteriormente visitado para atingir o nó atual. Assim sendo, os pontos do *Grid Maze* foram associados a nós de um grafo, numerados de 0 a 29 da seguinte forma.

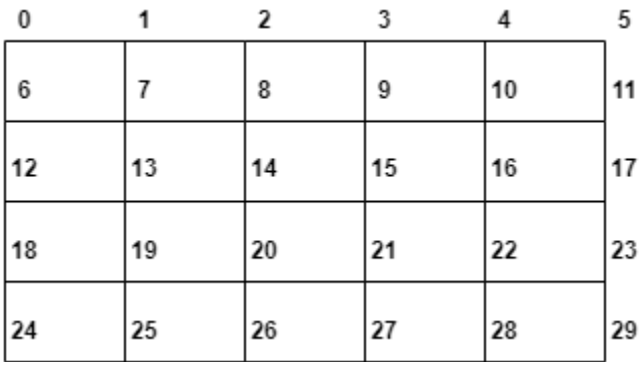

Figura 3 – Numeração dos nós do *Grid Maze*

A função *fillAdj()* é utlizada preencher a matriz *adj[][]* com as distâncias entre nós adjacentes. Como todos os nós são equidistantes este valor é sempre de 1, exceto nos casos onde dois nós não são adjacentes, onde toma o valor zero.

A função *Dijkstra()* (adaptada de https://www.programiz.com/dsa/dijkstra-algorithm) permite determinar a menor rota possível entre dois pontos de um grafo. Inicialmente, o custo para percorrer dois nós é mapeado na matriz *cost[][]* com a distância entre os nós, se adjacentes, ou infinito caso contrário.

Seguidamente, o vetor *distance[]* é preenchido com custo das distâncias entre todos os nós e o inicial. Em cada posição do vetor *pred[]* encontra-se o nó predecessor do nó atual, através do qual é garantido o menor caminho, definido inicialmente como o nó inicial. O vetor *visited[]* tem o valor 0 se o nó ainda não foi visitado ou 1 caso contrário.

Depois, são percorridos todos os nós do grafo, de modo a descobrir o nó que menos dista do inicial. Atualiza-se a distância mínima ao nó inicial, *min_distance*, com a menor distância do vetor *distance[]*, se o nó ainda não for visitado. O valor de *next_node* corresponde ao nó cuja distância ao inicial é a menor possível, logo passa a ser visitado uma vez que não existem caminhos mais rápidos.

```
for (i = 0; i < 30; i++) {
    if (distance[i] < min_distance && !visited[i]) {
        min_distance = distance[i];
        next_node = i;
    }
}
visited[next_node] = 1;
```

Seguidamente, são atualizados os vetores *distance[]* e *pred[]* se um nó não visitado puder ser acedido mais rapidamente através do nó que menos dista do inicial, *next_node*, ou seja, se existir um caminho entre nó atual e o inicial mais rápido. Se este for o caso, a distância corresponde à soma entre a distância do *next_node* e a distância entre este e o nó atual. Consequentemente, o predecessor do nó atual é atualizado.

```
for (i = 0; i < 30; i++) {
    if (!visited[i]) {
        if (min_distance + cost[next_node][i] < distance[i]) {
            distance[i] = min_distance + cost[next_node][i];
            pred[i] = next_node;
        }
    }
}
```

Este processo é repetido para todos os nós do grafo, sabendo-se no fim todas as rotas mais rápidas entre qualquer nó e o inicial. É então necessário identificar a rota entre o nó inicial e o final, sendo desenvolvida a função *findPath()*. O vetor dos predecessores dos nós são percorridos sucessivamente, começando pelo nó final, e adicionados ao vetor *path[]* até ser encontrado o nó inicial.

```
int curr_node = END_NODE;

path[0] = curr_node;
path_size = 1;

while (curr_node != start_node)
{
    path[path_size] = pred[curr_node];
    curr_node = path[path_size];
    path_size++;
}
```

Finalmente é necessário inverter o vetor *path[]* de modo a este começar no nó inicial e terminar no final.

Por fim, foi implementada a função *removeAdj(),* utilizada quando é detetado um objeto, para remover a adjacência de nós adjacentes ao obstáculo.

MÁQUINAS DE ESTADO

Devido à complexidade do projeto as máquinas de estado ilustradas seguidamente foram simplificadas de modo a facilitarem a sua leitura.

Para além disso, através da variável gridMaze_MODE, é possível alternar entre os diferentes modos de funcionamento do robô. Quando a variável toma o valor de 0, o robô encontra-se no modo de funcionamento normal, onde segue a linha de acordo com a regra da mão direita, sendo capaz de resolver labirintos sem loops. Com a variável igual a 1, percorre o Grid Maze de forma aleatória até eventualmente atingir a posição final. No modo 2, o Grid Maze é resolvido com o algoritmo Dijkstra.

MODO 0

Neste modo, as prioridades das máquinas de estado são as seguintes: *fsm_Obstacle*, *fsm_Junction*, *fsm_Line*.

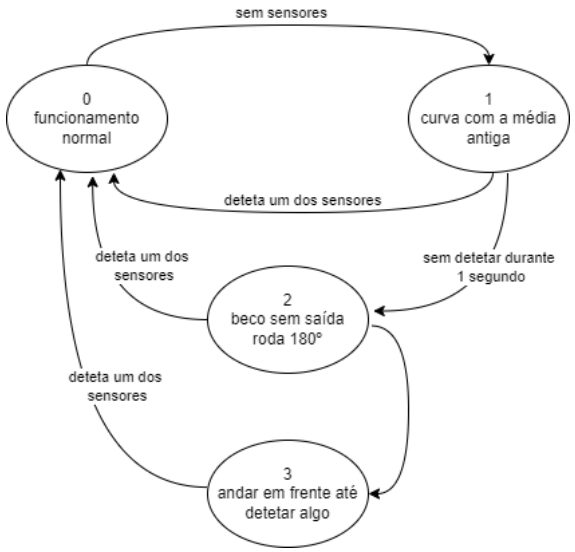

Figura 3 – *fsm_Line*

A máquina de estados *fsm_Line* é utilizada para ajustar a trajetória do robô com a variação da linha. O funcionamento normal do robô implica calcular a média ponderada naquele instante, calcular o valor da *speed1* e *speed2* com o algoritmo PID e aplicar essas velocidades ao modo F do *motorAction()*.

Nas curvas largas basta aplicar uma velocidade diferente nos motores, estando ambos a rodar no sentido dos ponteiros do relógio. Se o valor da média for menor ou igual a 1000 ou maior ou igual a 5000, existe uma curva apertada, logo, no estado de funcionamento normal, é necessário inverter o sentido de uma das rodas, ou seja, modo R ou L do *motorAction(),* respetivamente, de modo a realizar a curva sem se desviar de forma significativa da trajetória definida.

Se os sensores deixarem de detetar a linha, assume-se que está a realizar uma curva muito apertada. Em resposta, durante um segundo, o robô realiza um funcionamento semelhante ao funcionamento normal, mas com o valor da última média ponderada calculada. Isto permite ao robô voltar a detetar a linha, com trajetória anterior. Este estado também permite o correto funcionamento quando a linha é interrompida.

No caso de não voltar a encontrar a linha no espaço de tempo de 1 segundo, assume que encontrou um beco sem saída e roda 180° avançando até voltar a detetar um dos sensores.

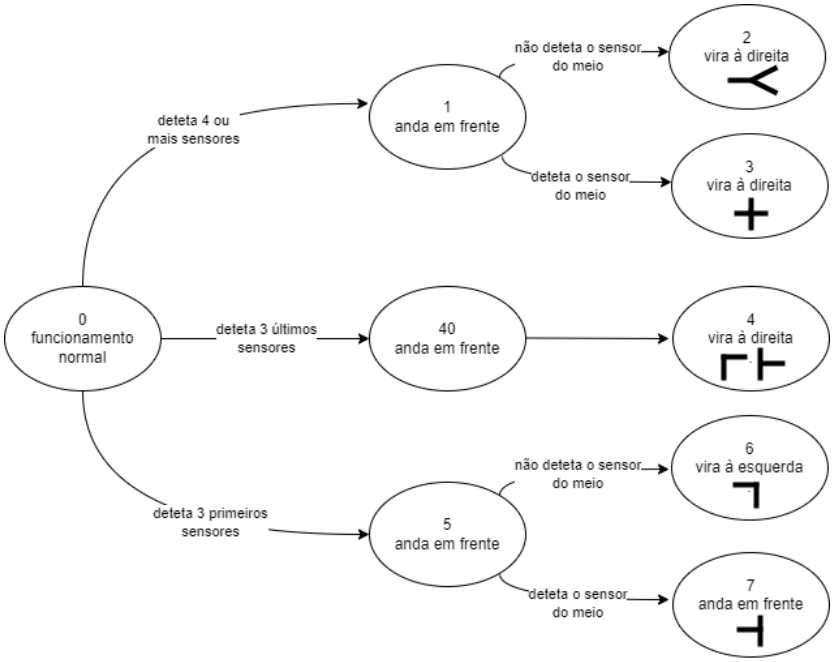

Figura 4 – *fsm_Junction*

A máquina de estados *fsm_Junction* serve para permitir ao robô distinguir os diferentes tipos de cruzamentos. A implementação desta máquina de estados define a prioridade das direções a seguir nos cruzamentos através da regra da mão direita. Sempre que possível vira-se à direita, senão segue-se em frente e, no caso de nenhum ser possível, vira-se à esquerda. Esta estratégia permite a resolução de labirintos simples onde não existam *loops*.

Tanto o estado 1 como o estado 5 são considerados estados de decisão onde o robô continua a mover-se em frente de modo a determinar o tipo de ação que deverá tomar tendo em conta o tipo de irregularidade que encontra. No estado 1, o Robô avança de modo a verificar se continua a detetar o sensor do meio e assim distinguir um cruzamento do tipo X do Y. O estado 5 permite distinguir se o robô deverá continuar em frente ou virar à esquerda.

O estado 40 existe para ajustar a posição final do robô quando existe uma curva de 90° à direita, de modo a acabar com a linha praticamente no centro dos sensores infravermelhos. Note-se que neste estado não é necessário verificar se existe a possibilidade de ir em frente, uma vez que o caminho à direita tem prioridade.

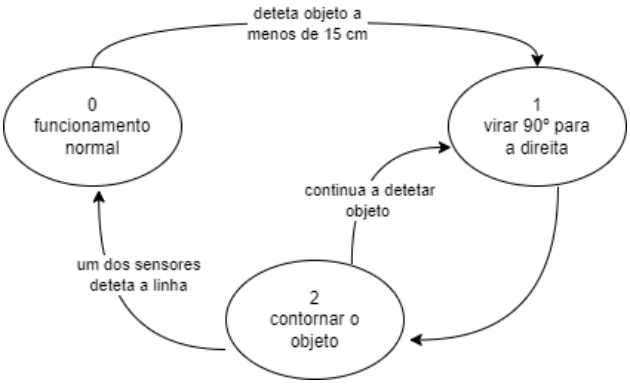

Figura 5 – *fsm_Obstacle*

A máquina de estados *fsm_Obstacle* é utlizada para evitar obstáculos. Quando é detetado um objeto a uma distância inferior a 15 centímetros, aplica-se o modo R do *motorAction()* durante um segundo, de modo a rodar o robô 90°. Consequentemente, o objeto é contornado numa trajetória circular, aplicando-se o modo F do *motorAction()* com a velocidade da roda interior a metade da roda exterior, até ser novamente detetada a linha. Se o objeto for novamente detetado o processo repete-se.

MODO 1

Neste modo, as prioridades das máquinas de estado são as seguintes: *fsm_GridMaze_Random_Obstacle*, *fsm_GridMaze_Random*.

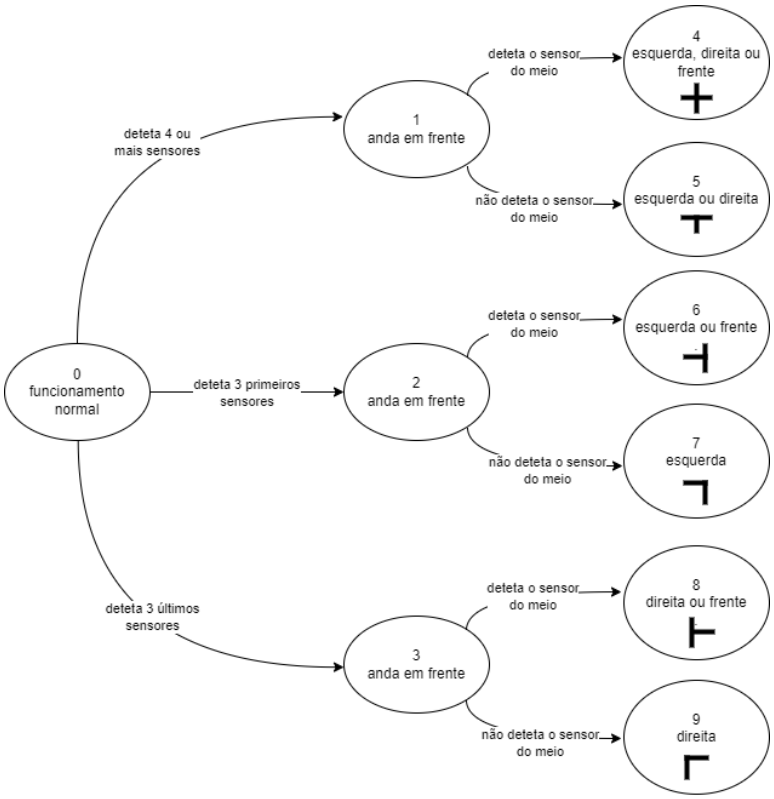

Figura 6 – *fsm_GridMaze_Random*

Nesta resolução, sempre que o robô deteta qualquer tipo de cruzamento é utilizada a função *random()* que escolhe um número entre 0 e 1 ou 0 e 2, dependendo do número de opções que o robô tem para escolher, seguindo o caminho correspondente ao número que foi selecionado.

A máquina de estados *fsm_GridMaze_Random,* semelhante a *fsm_Junction*, permite distinguir os vários tipos de cruzamentos e, assim, saber quantas opções de movimento o robô pode optar. Sempre que este vira à direita ou esquerda é guardado o movimento numa variável, *R* ou *L*, respetivamente.

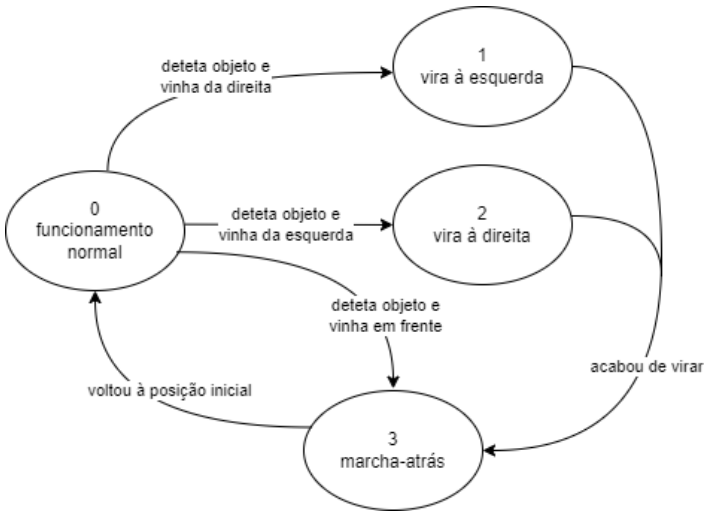

Figura 7 – *fsm_GridMaze_Random_Obstacle*

A deteção de objetos no labirinto é feita através da máquina de estados *fsm_GridMaze_Random_Obstacle*. Sempre que o robô deteta um objeto a uma distância menor que 10 centímetros, dependendo do último movimento que realizou, faz o inverso para voltar à posição inicial, e marcha-atrás durante um tempo definido para detetar novamente o cruzamento. De seguida, recomeça o processo da máquina de estados *fsm_GridMaze_Random* até encontrar um caminho que não tem nenhum obstáculo.

MODO 2

Neste modo, as prioridades das máquinas de estado são as seguintes: *fsm_GridMaze_Dijkstra*, *fsm_GridMaze_Dijkstra_FollowPath*, *fsm_GridMaze_Dijkstra_Orientation*

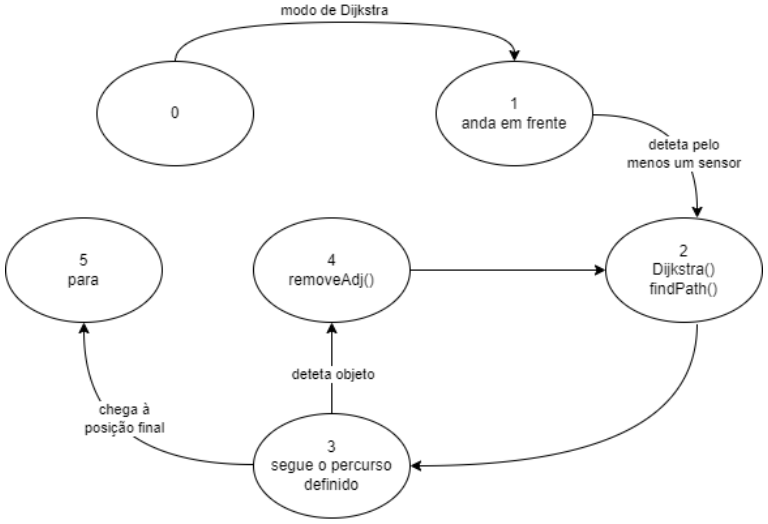

Figura 8 - *fsm_GridMaze_Dijkstra*

A máquina de estados *fsm_GridMaze_Dijkstra* é utlizada para resolver o *Grid Maze*. Os estados 0 e 1 são simplesmente utilizados para posicionar o robô em cima da sua posição inicial, onde deteta a linha. No estado 2 é calculado o caminho entre o nó inicial e final através das funções *Dijkstra()* e *findPath()*, descritas anteriormente. O estado 3 é utilizado para ativar a máquina de estados *fsm_GridMaze_Dijkstra_FollowPath* onde é seguido o caminho até a posição atual do robô corresponder ao nó final, onde o robô para.

Se durante o percurso for detetado um objeto, os caminhos adjacentes ao nó obstruído serão removidos através da função *removeAdj()* e recalculado um novo caminho no estado 2, com o nó inicial é igual à posição atual do robô. No entanto, devido a dificuldades na implementação, não foi possível implementar esta estratégia para lidar com a presença de obstáculos, pelo que no projeto final, o estado 4 foi removido.

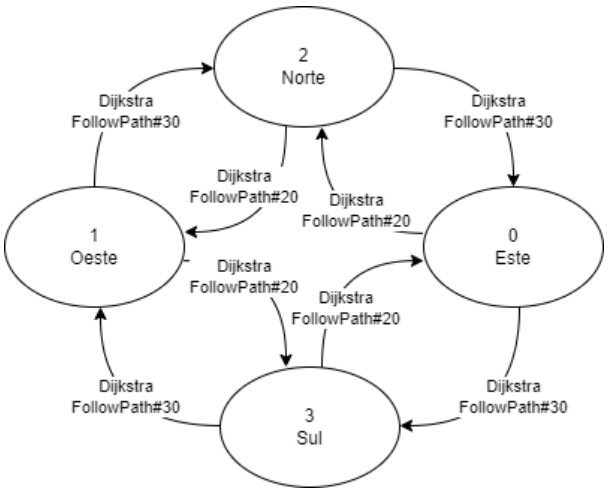

Figura 9 – *fsm_GridMaze_Dijkstra_Orientation*

A máquina de estados *fsm_GridMaze_Dijkstra_Orientation* é utilizada para se saber a orientação do robô no *Grid Maze,* tendo em conta o sentido da rotação indicado pelos estados 20 e 30 da máquina de estados *fsm_GridMaze_Dijkstra_FollowPath*. Note-se que o robô assume que é inicialmente colocado para a direção Este.

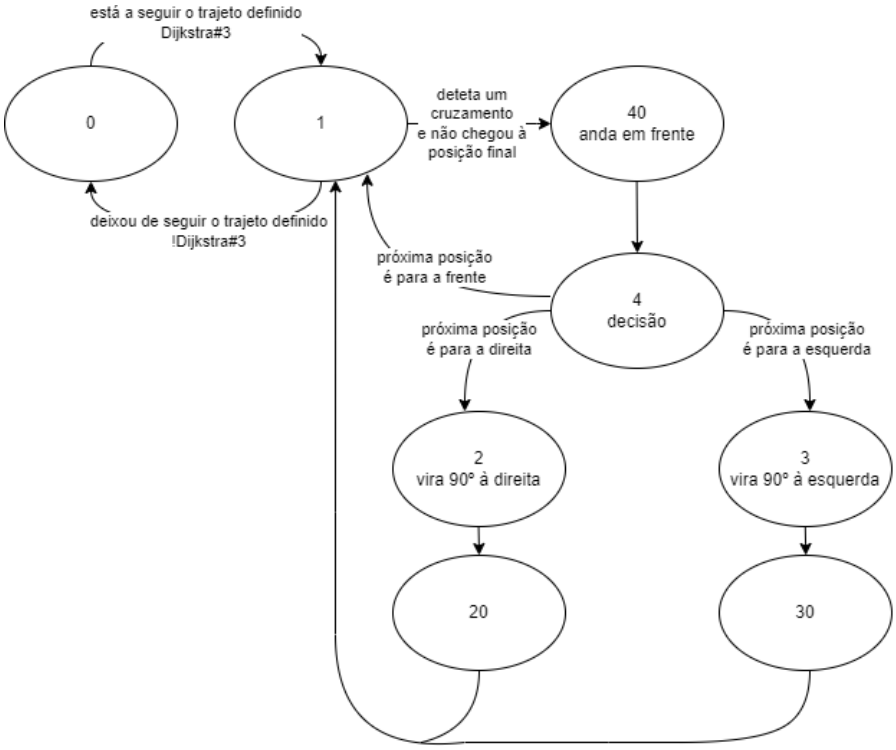

Figura 40 – *fsm_GridMaze_Dijkstra_FollowPath*

A máquina de estados *fsm_GridMaze_Dijkstra_FollowPath* é utilizada para seguir o caminho calculado em *fsm_GridMaze_Dijkstra* quando esta entra no estado 3. No estado 1, a *fsm_Line* está ativa no funcionamento normal, e impedida de evoluir, de modo ao robô seguir a linha e ajustar-se quando necessário.

Na transição do estado 1 para 40, é atualizada a posição atual do robô e incrementado o vetor *path[]* para o nó seguinte, tal que:

```
else if(fsm_GridMaze_Dijkstra_FollowPath.state == 1 && (in.IR1+in.IR2+in.IR3+in.IR4+in.IR5>=3) &&
curr_maze_pos != END_NODE){
      fsm_GridMaze_Dijkstra_FollowPath.new_state = 40;
      path_iterator++;
      curr_maze_pos = path[path_iterator];
}
```

A evolução do estado 4 para 2, onde o robô vira 90° à direita é definida pelas seguintes transições:

```
else if(fsm_GridMaze_Dijkstra_FollowPath.state == 4 && (
      (fsm_GridMaze_Dijkstra_Orientation.state == 2 && path[path_iterator+1] == curr_maze_pos + 1) ||
      (fsm_GridMaze_Dijkstra_Orientation.state == 0 && path[path_iterator+1] == curr_maze_pos + 6) ||
      (fsm_GridMaze_Dijkstra_Orientation.state == 1 && path[path_iterator+1] == curr_maze_pos - 6 ) ||
      (fsm_GridMaze_Dijkstra_Orientation.state == 3 && path[path_iterator+1] == curr_maze_pos - 1)))
  fsm_GridMaze_Dijkstra_FollowPath.new_state = 2;
```

Tendo em conta a numeração dos nós e a posição atual do robô, a sua orientação e o nó seguinte no caminho é decidido se o robô tem que virar à direita ou à esquerda. Por exemplo, se a posição atual for 0, a sua orientação Este e a posição seguinte for igual à atual mais 6, o nó seguinte está a Sul do atual e é necessário virar à direita. De um modo semelhante, se o robô estiver corretamente orientado para a posição seguinte ocorre a transição do estado 4 para 1, onde o robô segue em frente.

CONCLUSÃO

Este projeto permitiu o desenvolvimento de competências práticas associadas às boas práticas eletrónicas da montagem dos diversos componentes, assim como o aprofundamento de conhecimento associado à implementação de máquinas de estado capazes de controlar o robô do modo definido, com o auxílio de algoritmos de controlo e *path finding*.

O robô desenvolvido é capaz de seguir a trajetória de uma linha irregular, incluindo cruzamentos do tipo X e Y, curvas de 90° e interrupções na linha. Consegue ainda reagir à presença de obstáculos na sua trajetória, evitando ou contornando o obstáculo de acordo com o modo definido. Para além disso, é capaz de calcular a rota mais rápida entre dois pontos de um *Grid Maze*, tendo em conta a sua localização. No entanto, não foi possível implementar corretamente a deteção de obstáculos no *Grid Maze* com o algoritmo *Dijkstra.*

Apesar das dificuldades, considera-se que a implementação do robô foi um sucesso uma vez todos os objetivos definidos foram conseguidos.