Assignment 2 - RMI

Diogo Mendes (88801) e Raquel Pinto (92948)

Universidade de Aveiro

1 Introdução

Neste projeto, foi colocado um desafio complementar ao projeto anterior. Foi pedido para desenvolvermos um agente robótico capaz de navegar e de se localizar num labirinto desconhecido, utilizando o modelo de movimento, a distância aos obstáculos e a bússola. O GPS não está disponível, sendo que os motores, sensores de distância e a bússola têm ruido.

O agente tem que explorar um labirinto desconhecido a fim de mapear completamente o mapa onde se encontra. Ao mesmo tempo este precisa de localizar os *targets* colocados no labirinto. O número destes pode variar entre labirintos. Ao completar a tarefa de mapeamento, o agente deve regressar ao ponto de partida. Depois disto, o agente precisa de calcular um caminho fechado com um custo mínimo que permita visitar todos os *targets*, começando e terminando no *target* zero.

Deste modo desenvolveu-se uma maquina de estados que indica ao robô o próximo passo. Quando o robô chega ao meio da próxima célula, é calculado a próxima posição que o robô deve atingir e é guardada a vizinhança dele (incluindo os *targets*) para mais tarde ser impresso no mapa e no ficheiro do melhor caminho entre os *targets*.

Sendo este projeto uma continuação do projeto anterior, explicaremos primeiro o que foi feito anteriormente e a seguir o que se alterou para atingir os objetivos mencionados.

2 Projeto anterior

Foi nos pedido para que o robô consiga explorar um mapa desconhecido e no final imprimir num ficheiro o mapa que descobriu. É importante referir que escolheu-se colocar os sensores espaçados a cada 90^{o}, sendo que ficaram então a apontar para: 0^{o}, 90^{o}, -90^{o} e 180^{o}.

Usou-se uma máquina de estados com 7 estados (*GA*, *RL*, *RR*, *INV*, *WAIT*, *COLLISION* e *END*) que diz ao robô qual é o próximo passo. Se o robô estiver no estado *WAIT* significa que a simulação ainda não começou, o robô está a espera que o utilizador a inicie. Se estiver no estado *GA* ele anda em frente, no estado *RL* ou *RR* ele vira para a esquerda e para a direita, respetivamente. Se estiver no estado *INV* significa que o robô terá de inverter. Se o robo colidir com alguma parede entra no estado *COLLISION* onde atualiza as coordenadas para

onde ir para as ultimas onde já esteve e passa para o estado *INV* e finalmente quando o robô está no estado *END* termina o desafio.

A cada ciclo, antes de entrar nos *case* da máquina de estados é feita uma análise do melhor estado para o robô (*função Estados()*) que através do valor da bússola atual (*compass*) e do valor da bússola objetivo (*compass_goal*) consegue definir o próximo estado. Note-se que as coordenadas objetivo são definidas noutra função. Desta forma, esta função consegue analisar se o robô está na direção correta para andar em frente ou se precisa de rodar mais para um lado.

Nos estados *GA*, *RL* e *RR*, é avaliado se o robô atingiu o seu objetivo (deslocar-se até ao meio da próxima célula) através da função *targetReached()* dando sempre uma margem de mais ou menos 0,15 em relação ao centro. Também é avaliado se o robô se encontra numa situação de beco, o que vai ser explicado mais à frente.

Se a função *targetReached()* nos devolver que o robô atingiu o seu objetivo é fornecido às rodas a velocidade (0,0) e, de seguida, chamada a função *mappingDecode()*, se ele ainda não tiver atingido o objetivo é chamada uma das funções *goAhead()*, *goLeft()*, *goRight()* consoante o estado onde ele se encontra (*GA*, *RL* e *RR*, respetivamente) que lhe permite andar em frente, rodar para a esquerda ou para a direita. Estas funções definem a velocidade a aplicar em cada roda, utilizando o método abordado na aula de controlo, *PID*. No caso destas funções, a abordagem utilizada é apenas proporcional, ou seja, multiplica-se uma constante pelo erro, em x ou em y, consoante o eixo do movimento, para obter a velocidade de cada roda. Os valores para constantes foram obtidos por tentativa e erro.

Quanto ao *mappingDecode()*, esta é uma das principais funções. Para compreender esta função é importante perceber que ela apenas é chamada quando o robô se encontra no centro de uma célula que era a célula objetivo!

```
void mappingDecode(){
    //guardar o valor das coordenadas atuais
    //na lista coordsAntigas
    coordsAntigas.add(vetor atual);

    if(target){
        //inserir o target na posicao do ground do array
        objetivos[ground] = vetor atual;
    }
    //mapear a vizinhan a com os sensores
    coord[x vetor atual][y vetor atual] = ``X'' || ``|'' || ``-'';
    //conforme a situacao

    //verificar posicoes visitaveis e adicionalas
    //lista com todas as posicoes visitaveis
    //que ainda nao foram visitadas
    visitaveis.add(vetorposicaovizitavel);
    //lista so com as coordenadas possiveis a volta do robo
```

```
    localViz.add(vetorposicaovizitavel);

    if(localViz.isEmpty()){//se o robo estiver num beco
        aStar(); //calcular o melhor caminho
                 //para a proxima posicao visitavel
        runCaminho(); //percorrer o caminho dado pelo aStar()
    }
    atualizar a proxima posicao a ser atingida pelo robo
    atualizar o compass_goal
    writeMap();
}
```

Listing 1.1. MappingDecode() pseudocódigo

Como se pode ver na Listing 1.1, a função começa por guardar o valor das coordenadas atuais na *LinkedList coordsAntigas*, que contém todas as coordenadas por onde já passou. E retira o vetor atual da *LinkedList visitaveis*.

A seguir verifica se o robô se encontra num *target*. Se isto se verificar, ele insere no array *objetivos* no índice correspondente ao valor da ordem do target (dada pela variavel *ground*) as coordenadas desse, por exemplo o *target* 4 vai para o índice 4, é importante ter esta ordem correta para calcular corretamente o caminho no final.

Posteriormente, utilizando os valores obtidos pelos sensores, o robô faz um mapeamento da sua vizinhança e guarda num array de 2 dimensões (*coords*) os dados obtidos (se é parede, espaço livre ou um *target*) para escrever o mapa num ficheiro. Usando também esta informação a função adiciona a um *Set* temporário, as coordenadas para vizinhas de movimento possível (sem parede) e nas quais o robô ainda não esteve.

Caso, este Set seja vazio, ou seja, não existem movimentos possíveis para posições na vizinhança imediata onde ele ainda não esteve é chamada uma função *setCaminho* que vai criar um caminho, usando o algoritmo de pesquisa A estrela, para as coordenadas mais próximas marcadas na lista visitáveis.

Caso o Set contenha valores, vai ser calculada a próxima posição e o próximo valor de bússola objetivo (*compass_goal*) do robô.

A função *setCaminho()* corre o código A estrela explicado abaixo com o vetor atual como raiz e com um dos vetores da lista visitaveis. Após calcular esse caminho que é guardado na *LinkedList caminho* é chamada a função *runCaminho()* que vai definir o primeiro valor do caminho como *next* e alterar a bússola do objetivo (*compass_goal*) de acordo com isto. Sendo que no fim de cada execução é eliminado o primeiro valor do caminho.

Enquanto existirem valores no caminho a função *MappingDecode* vai continuar a chamar a *runCaminho()*.

Relativamente ao algoritmo de busca, usou-se o link mensionado na secção da bibliografia como base para a classe *Node*, para a classe *aStar* e para as funções a elas associadas.

Naturalmente, que o código foi apenas usado como base, tendo sido bastante alterado para adaptar-se ao nosso desafio. Algumas das principais alterações relacionam-se com a adição de um atributo para guardar o vetor correspondente ao nó, a adição de uma forma de *deep copy* dos filhos de um vetor quando se abre o nó, pois estavam a ser perdidos valores.

A função *printPath()* apesar do nome, foi alterada para devolver uma *LinkedList* com o caminho calculado.

Para além, destas funções principais, temos várias funções e classes auxiliares, entre as quais foi criada uma classe vetor (com seus atributos o X, o Y e os filhos, que neste caso são os vizinhos para onde o robô pode ir a partir desse vetor), e várias listas.

Criou-se um vetor chamado *next* que armazena as coordenadas seguintes, ou seja, as coordenadas para onde o robô se deve deslocar. Criou-se também uma lista chamada *coordsAntigas* que guarda as coordenadas dos centros das células onde o robô passou, uma lista chamada *visitaveis* que guarda as coordenadas dos centros das células que já foram marcadas como passagens ('X') e ainda não foram visitadas, e uma lista chamada *caminho* que contém o caminho em situações de beco.

Neste projeto tivemos de analisar quatro casos:

O primeiro caso, ocorre quando o robô tem paredes à esquerda e à direita, como regra geral, ele não anda em recuo as coordenas atrás dele serão parte do *coordsAntigas*, pelo que o *next* corresponde às coordenadas diretamente à frente do robô e o estado, normalmente, é o *GA*.

No segundo caso, o robô chega a uma curva, em que tem parede à frente e num dos lados o que significa que o seu *next* vai ser num dos seus lados e por isso terá de alterar o eixo do movimento. Na função *MappingDecode* não só vão ser calculadas as coordenadas seguintes como o *compass_goal* é alterado para um valor correto. Assim, a máquina de estados entrará no estado *RL* ou *RR* conforme tiver que rodar à esquerda ou à direita, respetivamente. E estes estados, com as funções de movimento correspondentes, farão o robô rodar até o *compass* ser igual ao *compass_goal*. Finalmente, depois da bússola alinhar, o estado *GA* fará com que ele ande em frente até ao objetivo.

O terceiro caso é semelhante, em parte, ao segundo, mas só com uma parede a ser detetada, ou sem parede de todo. Ou seja, quando o robô alcança um cruzamento, definiu-se cruzamento como a situação em que o robô pode ir para 2 ou 3 posições onde ainda não esteve. É de salientar que, caso o robô tenha estado previamente numa das posições, ou seja, uma das posições possíveis sem voltar para trás já foi visitada anteriormente, será considerado como uma parede. No caso de, só sobrar uma posição possível tratar-se-á de um dos casos anteriores consoante essa posição for frontal ou lateral.

Neste caso de cruzamento, a função *mappingDecode()* seleciona a primeira coordenada que guardou no Set de posições com movimentos possíveis como a posição objetivo, *next*, e guarda as outras na lista *visitaveis*.

Por último, no quarto caso o robô está num beco, esta situação foi definida como uma posição sem movimentos possíveis para coordenadas à sua volta não visitadas. Tal situação pode ocorrer quando o robô chega a um cruzamento onde todas as coordenadas à sua volta estão na lista *coordsAntigas*. Um exemplo simples, ocorre quando, o robô tem parede à frente, à esquerda e à direita (um beco sem saída) em que a única posição para onde se pode mover é aquela de onde veio, ou seja, já foi visitada.

Neste caso, a próxima posição que o robô quer atingir (*next*) vai ser dada pelo último vetor inserido na lista *visitaveis*. Para o robô conseguir atingir esse ponto, utilizou-se uma pesquisa em A estrela que devolve o caminho ótimo (de menor custo) entre a posição atual e a posição *next* (último vetor inserido na lista *visitaveis*). As coordenadas deste caminho são inseridas na lista caminho e a função *runCaminho()* é chamada para movimentar o robô ao longo desse caminho.

Este desafio termina quando o robô tiver percorrido o mapa todo (ou seja, quando a lista visitaveis estiver vazia) ou quando o tempo do desafio terminar.

Após percorrer o mapa todo, o programa vai calcular o melhor caminho para percorrer todos os *targets* guardando-o numa LiskedList.

Finalmente, o estado *END* ocorre em duas situações. A primeira, quando o robô tiver percorrido o mapa todo. A segunda, quando o número de ciclos for maior que 4990, isto serve como precaução caso ocorra algum erro no código que impeça o mapeamento total. Neste estado, é escrito uma ultima vez o mapa guardado no *array coords* num ficheiro de texto e o melhor caminho calculado para percorrer os *targets*. Quer o simulador quer o programa são terminados.

3 Projeto atual - Localização

3.1 Atualizar X e Y pelos motores

No projeto atual, como não temos acesso ao GPS, tivemos de calcular a posição atual do robô com o movimento das rodas. Para isso sempre que fazemos um *cif.DriveMotors()* atualizamos a posição atual do robô (função *updateAll*) utilizando para tal as equações fornecidas no enunciado do projeto, garantindo sempre, que esta função é chamada a cada ciclo.

Assim nesta função calculou-se e atualizou-se o valor atual das coordenadas, das coordenadas anteriores, da bússola anterior e dos *outputs* dos motores atuais e anteriores. Esta atualização ou é direta (e.g. *compassLast=compass*) ou é feita recorrendo às equações 1, 2, 3 e 4.

$$out_t = \frac{in_i + out_{t-1}}{2} \tag{1}$$

onde out_t é aplicado no tempo t, ou seja out_{t-1} é aplicado no instante de tempo t-1 (ciclo anterior).

$$x_t = x_{t-1} + lin * cos(\theta_{t-1}) \tag{2}$$

$$y_t = y_{t-1} + lin * sen(\theta_{t-1}) \tag{3}$$

$$lin = \frac{out_t^l + out_t^r}{2} \tag{4}$$

É possível calcular um valor aproximado para o x, y, $outl$ e $outr$. Em que $outl$ e $outr$ correspondem aos valores introduzidos no motor do lado esquerdo e direito, respectivamente.

3.2 Correção usando Sensores

Visto que os motores têm ruído é necessário efetuar uma correção ao X e Y calculados utilizando os sensores. Na mesma função referida anteriormente (*updateAll*) calculou-se os valores atuais do x e do y utilizando-se os valores obtidos pelos sensores de obstáculos.

Para efetuar esta correção, usou-se os valores obtidos pelos sensores colocados a $0^{\underline{o}}$ e $180^{\underline{o}}$ (frente e trás). Utilizando uma função auxiliar e o X calculado pelos motores calculou-se a posição da parede da frente e da parede de trás, tendo em conta para onde o robô está a apontar, sendo que se tem pelos sensores a distância para essas paredes.

Retirou-se à posição da parede da frente a distância calculada entre o robô e essa parede, retirando-se também o raio do robô (equação 5). Por sua vez, à parede de trás é somada a distância assim como o raio do robô (equação 6).

$$xy = parede - ((dist + 0.5) * nivel) \tag{5}$$

$$xy = parede + ((dist + 0.5) * nivel) \tag{6}$$

Sendo que nivel é uma variável cujo valor é 1 quando o robô se encontra a apontar para um eixo no sentido positivo e -1 quando se encontra a apontar para um sentido negativo.

A equação 5 é válida para o cálculo do X ou Y usando o sensor da frente, e por sua vez, a equação 6 é utilizada para o cálculo do X ou Y usando o sensor de trás. Estas equações foram alcançadas após análise e desenho das várias situações possíveis e síntese das mesmas.

Utilizou-se a função *Eixo()* para decidir se o resultado das equações anteriores é referente ao X ou Y.

É tido em conta a existência ou não de parede, sendo que no caso de não existir não ocorre qualquer correção. No caso de existir apenas um sensor detetar uma parede é apenas usado a coordenada calculada com o valor deste sensor, enquanto que se existirem paredes em ambos os sensores é feita uma média aritmética entre as coordenadas calculadas usando os valores de ambos.

Implementou-se uma correção do X ou Y, dependendo do eixo do movimento, utilizando os sensores laterais (90^{o} e -90^{o}) de forma similar à apresentada para os sensores frontal e traseiro.

Foi tido em conta que o erro das coordenadas obtidas pelos motores e pelos sensores são diferentes, e por isso, foi aplicada uma média ponderada entre ambos para obter as coordenadas finais com um peso de 80% para as coordenadas obtidas pelos motores e 20% para as coordenadas obtidas pelos sensores.

No entanto, como vai ser possível observar na secção 4 devido aos piores resultados que se obteu utilizando as correções com sensores esta parte do código foi comentada sendo que para a entrega final apenas realiza as atualizações pelos motores.

4 Discução e resultados

A nível de resultados, após executar o programa completo com todas as funções apresentadas na seção anterior. Obtiveram-se os seguintes resultados.

- No caso de ocorrer a correção com sensores apenas quando se encontra no centro de uma célula.
  - Tempo médio para a primeira colisão: 1400 ciclos
  - Número médio de colisões: 100 colisões
  - Tempo para terminar o mapeamento todo: NaN - nunca termina
  - Mapa no final: 1

Fig. 1. Mapa com todos os sensores, todas as funções e correção nos centros

– No caso de ocorrer sempre a correção.
  - Tempo médio para a primeira colisão: 400 ciclos
  - Número médio de colisões: 160 colisões
  - Tempo para terminar o mapeamento todo: NaN - nunca termina
  - Mapa no final: 2

Fig. 2. Mapa com todos os sensores, todas as funções e correção constante

Apesar de, às vezes este modelo levar a um bom mapeamento, uma das principais razões para este modelo ter sido abandonado é a sua instabilidade. Muito frequentemente o agente colide com uma parede e entra num ciclo vicioso de colisões acabando por não mapear mais o mapa.

Agora analisemos o caso em que retirasse a correção usando os sensores laterais e usasse apenas o frontal e traseiro.

– No caso de ocorrer a correção com sensores apenas quando se encontra no centro de uma célula.

- Tempo médio para a primeira colisão: 3000 ciclos
- Número médio de colisões: 120 colisões
- Tempo para terminar o mapeamento todo: 4600 ciclos (quando consegue terminar)
- Mapa no final: 3

Fig. 3. Mapa com 2 sensores e todas as funções de correção nos centros

– No caso de ocorrer sempre a correção.

- Tempo médio para a primeira colisão: 1100 ciclos
- Número médio de colisões: 160 colisões
- Tempo para terminar o mapeamento todo: 4300 ciclos (quando consegue terminar)
- Mapa no final: 4

Fig. 4. Mapa com 2 sensores e todas as funções com correção constante

É de realçar que em todos os casos anteriores a norma é não conseguir terminar o mapeamento por ficar preso num ciclo de colisões, daí ter valores médios de colisões tão elevados. No entanto, nas raras vezes em que de facto consegue terminar o mapeamento ou não ocorrem colisões de todo ou ocorrem menos de 10.

Finalmentemte, no caso, referido anteriormente, de não efetuarmos correção ao X e Y utilizando os sensores, obtemos os seguintes resultados:

- Tempo médio para a primeira colisão: 1100 ciclos
- Número médio de colisões: 160 colisões
- Tempo para terminar o mapeamento todo: 4300 ciclos (quando consegue terminar)
- Mapa no final: 5 e 6

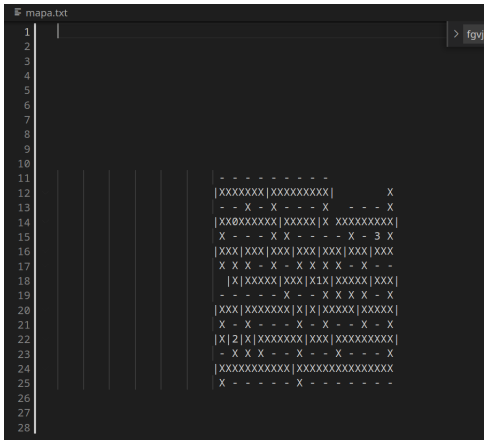

Fig. 5. Mapa apenas usando os motores.

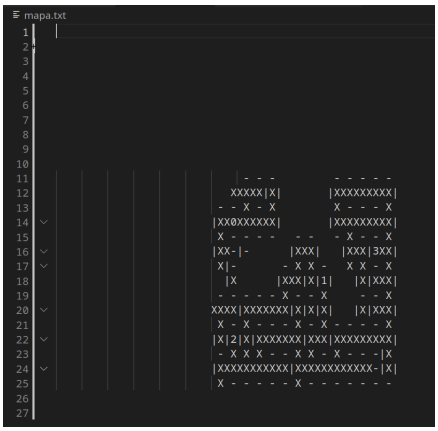

Fig. 6. Outro mapa apenas usando os motores.

Como podemos ver, por comparação das figuras e dos resultados obtidos concluíu-se que as funções de calculo do X e do Y com os sensores têm piores resultados comparativamente ao código só com atualização do X e Y pelos motores. Para além dos resultados serem melhores o facto de serem mais estáveis (número de vezes em que ocorrem deadlocks por colisões infinitas é muito menor).

5 Conclusão

Concluindo, neste projeto tentou-se de várias maneiras resolver o problema apresentado. Era de esperar que se tivesse melhores resultados com melhor desempenho utilizando métodos de correção das coordenadas para alem do cálculo do x e do y através dos motores, tal não se verificou na prática. Por isso foi decidido que para esta entrega iria ser feito o calculo do x e do y apenas com as equações 1, 2 , 3 e 4.

Como trabalho futuro, sugerimos entender-se o porquê das funções para correção do X e Y usando sensores, não corrigirem mas sim piorarem os valores obtidos.

6 Bibliografia

https://stackabuse.com/graphs-in-java-a-star-algorithm/