

# RoboCode

André Figueiredo, Luís Ferreira, Pedro Machado, and Rafael Lourenço

Universidade do Minho, Departamento de Informática, 4710-057 Braga, Portugal

Perfil de Sistemas Inteligentes

Sistemas Autónomos, Trabalho Prático Nº1, Grupo 7

11 de Abril de 2021

e-mail: {a84807, a86265, a83719, a86266}@alunos.uminho.pt

**Resumo** Neste relatório será apresentando o trabalho prático de Sistemas Autónomos que aborda as três fases solicitadas, desenvolvidas no âmbito do jogo RoboCode, que permite, por exemplo, simular batalhas entre *robots* criados. Assim, iremos descrever a nossa metodologia para cada uma destas fases, explicando a abordagem tomada na primeira fase para circum-navegar os obstáculos, os comportamentos sociais da segunda fase e as estratégias e técnicas utilizadas para a competição da terceira fase, fornecendo sempre algum contexto nas decisões tomadas e no que se basearam. Por último, será feita uma análise dos resultados para a primeira e terceira fases onde os nossos *robots* competiram (tentando percorrer o caminho da forma mais eficiente) e batalharam contra outras equipas desenvolvidas pelos nossos colegas, sendo discutidas algumas das dificuldades, assim como futuras possíveis melhorias.

**Keywords:** Java · Robot · RoboCode · Autonomous Systems · Intelligent Agents · Multi-Agent Systems

## 1 Introdução

No âmbito da unidade curricular de Sistemas Autónomos foi-nos proposto o desenvolvimento de *robots* autónomos que permitam a simulação de comportamentos individuais, de grupo e sociais, no jogo RoboCode.

O projeto está dividido em três fases distintas, sendo que, na primeira fase, o objetivo é definir a rota mais eficiente, isto é, o caminho mais curto que um *robot* tem de percorrer para contornar certos obstáculos, e desenvolver um odómetro, para medir a distância percorrida pelo *robot* (e poder saber qual a mais curta). Posteriormente, na segunda fase, é pretendida a cooperação entre os agentes, ou seja, os *robots*, levando, assim, a diversos comportamentos de grupo. Por último, a terceira fase consiste na criação de um ambiente de competição entre os agentes, cujo objetivo é ganhar várias batalhas para vencer competições. Para isso, será construída uma equipa que apresentará várias estratégias de combate, onde todos os agentes comunicarão entre si com a mesma finalidade.

Em suma, o principal objetivo deste projeto passa pela programação de vários tipos de *robots*, tendo como base três classes já implementadas - AdvancedRobot, TeamRobot (extensão da classe AdvancedRobot) e Droid - sendo, para tal, desenvolvidas várias estratégias para os *robots* atingirem um dado objetivo, através da cooperação, execução de tarefas em grupo ou tomada de decisão em equipa/por um líder.

Deste modo, várias funcionalidades, como comunicação através de mensagens e aumento da “vida” do líder, passam a ter de ser suportadas em equipas (TeamRobot).

## 2 Fase 1

A primeira fase do projeto consiste no desenvolvimento de um odómetro para a medição da distância percorrida por um *robot* em cada *round* de uma batalha. Após o seu desenvolvimento, pretende-se aplicá-lo para poder efetuar uma comparação da distância do nosso odómetro com a distância do odómetro que nos foi fornecido pela equipa docente.

De seguida, foi também proposta a implementação de técnicas de circum-navegação de três obstáculos por um *robot* a implementar. Este *robot*, denominado por *BeastMaster64*, tem de começar o contorno dos obstáculos na posição (18, 18), circular no sentido dos ponteiros do relógio e voltar à posição inicial. Para esta circum-navegação, pretende-se, então, a minimização do percurso efetuado, ou seja, obter o valor mais próximo possível do perímetro do poliedro.

Desta seguida, iremos demonstrar as abordagens efetuadas, de forma a se obter o melhor resultado possível.

### 2.1 Movimentação

Para permitir que o *robot* possa percorrer o trajeto definido, é necessário ter uma função que permite deslocar o *robot* para uma dada posição. Assim, é necessário calcular a distância a percorrer e o ângulo que permite alinhar o *robot* com trajeto em linha reta.

Para a distância, trata-se de um cálculo simples, sendo apenas necessária a fórmula da distância euclidiana:

$$d = \sqrt{\Delta x^2 + \Delta y^2}$$

$\Delta x$  toma o valor da subtração da coordenada  $x$  da posição pretendida com a coordenada  $x$  atual do *robot*. Para  $\Delta y$ , o cálculo é análogo.

```
double dy = target_y - this.getY();  
double dx = target_x - this.getX();  
double distance = Math.sqrt(dx * dx + dy * dy);
```

Para calcular o ângulo, é necessário recorrer às seguintes fórmulas trigonométricas:

$$\tan \theta = \frac{\text{opposite}}{\text{adjacent}} \quad \text{e} \quad \theta = \tan^{-1} \left( \frac{\text{opposite}}{\text{adjacent}} \right)$$

Assim sendo, é necessário identificar os catetos adjacente e oposto, ou seja,  $\Delta x$  e  $\Delta y$ , respetivamente. Então, é apenas necessário calcular o arco-tangente da divisão de  $\Delta y$  por  $\Delta x$ , todavia esta implementação retorna um ângulo entre  $-\frac{\pi}{2}$  e  $\frac{\pi}{2}$ . Para contornar essa situação, recorremos à função do package *math atan2*, que permite calcular o arco tangente, retornando um ângulo entre  $-\pi$  e  $\pi$ .

O resultado desta função corresponde ao ângulo a que a posição pretendida está em relação à posição atual do *robot*. Assim, resta apenas calcular a diferença entre o ângulo calculado e o da orientação atual.

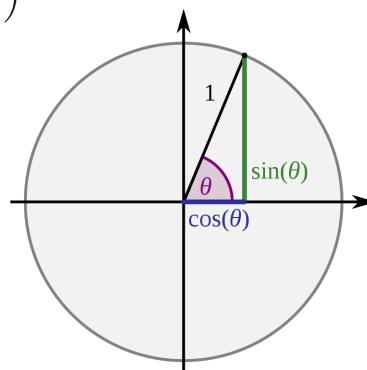


Figura 1: Círculo Trigonométrico

```
double turnAngle = (Math.toDegrees(Math.atan2(dx, dy)) -  
    this.getGunHeading() + 360) % 360;
```

Depois de ter o ângulo calculado, apenas é verificado se este é maior ou menor ou igual a 180°, para garantir que o *robot* nunca vira mais do que esse ângulo, tornando a ação fluída.

```
if(turnAngle <= 180) turnRight(turnAngle);  
else turnLeft(360 - turnAngle);
```

Depois de orientar o *robot*, resta ordená-lo a avançar a distância necessária.

```
this.ahead(distance);
```

## 2.2 Odómetro implementado

Primeiramente, o nosso odómetro verifica, constantemente, se o *robot* já se encontra na posição inicial prevista para a medição da distância percorrida, sendo que, neste caso, é o canto inferior esquerdo do mapa, dado pelas coordenadas (18, 18). Caso este se encontre nessa posição, mas não esteja pronto para a corrida, é porque se trata do início desta, pelo que se pode começar a adicionar as posições em que o *robot* se encontra ao seu histórico.

```
if(position.distance(initCoords, initCoords) < 0.0001 &&  
!this.isReady)  
    this.isReady = true;
```

De seguida, inserem-se as posições do *robot* à medida que a função é invocada, de forma a que, no final da circum-navegação, se possa calcular a distância total percorrida.

```
else if(position.distance(initCoords, initCoords) > 0.0001 &&  
this.isRacing)  
    this.history.add(position);
```

Por último, verifica-se se o *robot* chegou à posição da qual partiu para a corrida, sendo “sinalizado” que este terminou a circum-navegação através da variável `finished` com o valor `true`.

```
else if(position.distance(initCoords, initCoords) < 0.0001 &&  
this.isRacing)  
{  
    this.finished = true;  
    this.history.add(position);  
}
```

Esta distância é dada pela soma das várias distâncias do histórico de pontos por onde este *robot* passou. Todavia, é de realçar que esta pode não corresponder à distância “real” que o *robot* percorre, pois a implementação do odómetro<sup>1</sup> permite que a existência de curvas do percurso faça com que a adição dos pontos do caminho percorrido resulte em cortes/atalhos do trajeto.

---

<sup>1</sup> Esta é feita através de eventos no método `test` disponibilizado pela classe `Condition` do RoboCode.

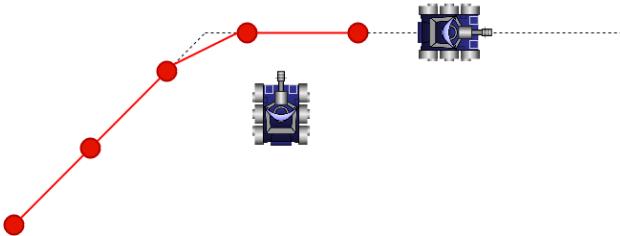


Figura 2: Corte da distância real percorrida pelos pontos do histórico do odômetro

Em comparação com o odômetro fornecido pela equipa docente, o nosso indica valores mais próximos dos reais, sendo mais fiável para o cálculo da eficiência do *robot* relativamente ao perímetro do poliedro formado pelos obstáculos. Esta diferença entre os dois odômetros poderá dever-se à conversão das posições inicial e final dos *robots* para inteiros por parte do odômetro fornecido pela equipa docente, visto que este *cast* pode traduzir-se numa redução do trajeto calculado.

### 2.3 Robot implementado

Para esta primeira fase foi, também, necessário desenvolver um *robot* capaz de contornar outros três *robots*, denominados por *RockQuads*, que, no início da batalha, calculam as suas posições nos respetivos quadrantes e dirigem-se para as mesmas.

Primeiramente, o nosso *robot* tem como objetivo chegar à posição inicial, de onde vai começar a circum-navegação dos *robots*, recalculando a rota sempre que embate num outro *robot*. Como, para os cálculos, estamos a tratar de *doubles* e arredondamentos de variáveis (como o  $\pi$ ), é necessário criar um delta de proximidade às coordenadas iniciais, usando, normalmente, o valor 0.001.

```
while (Math.abs(this.getX() - initCoords) > 0.001 ||  
    Math.abs(this.getY() - initCoords) > 0.001)  
    move(initCoords, initCoords);
```

Após esta etapa, é efetuado o cálculo de toda a rota que o *robot* vai tomar, sendo necessário fazer *scan* dos três obstáculos. Visto que, ao começar o *scan*, os outros *robots* podem, ainda, não estar nas suas posições<sup>2</sup>, foram tidos em consideração vários elementos para a erradicação de *bugs* provenientes destes problemas.

Assim, foram criadas duas estruturas de três elementos cada para serem colocadas as posições dos *robots*/obstáculos que foram alvo de *scan*. Uma das estruturas (a principal) simboliza o último *scan* de 360° e a outra simboliza o mais recente *scan* executado. No final de cada 360° é verificado se as duas estruturas são iguais e, só em caso positivo, é que se pode ter a certeza de que os *robots* se encontram na sua posição calculada. Para evitar colocar duas vezes o mesmo *robot* na estrutura, são apenas adicionados os *robots* que se encontram parados e nunca são adicionadas duas posições iguais.

No final desta fase são categorizados os obstáculos, como *northwest*, *northeast* e *southeast*, que representam os *robots* dos quadrantes 2, 1 e 4, respetivamente. Para este fim, são tidas em conta as coordenadas *x* e *y*, sendo que o *robot* com o *x* inferior é caracterizado como *northwest* e o com o *y* inferior é caracterizado como *southeast*, deixando o restante ponto para o *northeast*.

---

<sup>2</sup> Isto pode-se dever ao facto dos *robots* “nascerem” longe das suas posições ou por embaterem noutros *robots* ou de, por vezes, serem feitos vários *scans* seguidos ao mesmo *robot*.

Depois da categorização, segue-se o cálculo do caminho a ser percorrido, ou seja, da circum-navegação. Todos os pontos calculados são colocados numa estrutura para, posteriormente, serem percorridos de forma sequencial pelo nosso *robot*. É de ressalvar que o caminho obtido é o mais eficiente.

O cálculo e a execução da circum-navegação serão melhor explicados, com o auxílio de imagens, na secção 2.4.

Por fim, ao terminar o trajeto, são apresentadas todas as informações que se consideram úteis, isto é, a distância calculada pelo *Odometer* fornecido pelos docentes, pelo *Odometer* desenvolvido por nós e a distância “real” que é adicionada sempre que é efetuado um movimento. É, também, calculado o perímetro entre os *RockQuads* e a origem e a eficiência do percurso efetuado, tendo em conta os dois *Odometers*, a distância “real” e o perímetro calculado.

## 2.4 Circum-Navegação

A implementação da circum-navegação foi aquela que mais debate gerou, com vista a cumprir o objetivo de tentar obter o caminho mais eficiente para o contorno dos obstáculos.

Primeiramente, foram tidos em consideração os pontos que o *robot* teria de passar, de modo a contornar o percurso da forma mais eficaz. Após encontrar estes pontos foi, então, traçada uma linha entre eles, visto que o caminho mais perto entre dois pontos é sempre uma linha reta. Levou-se, ainda, em conta que cada *robot* tem uma *hitbox* de  $36 \times 36$  e que esta é independente do seu *heading*, ou seja, não é necessário ter em atenção a direção do *robot* para o contornar. Assim sendo, chegou-se à conclusão de que os pontos mais eficientes para contornar um *robot* são os seus vértices, contudo é preciso considerar, não só a *hitbox* do obstáculo, como também a do nosso *robot*. Deste modo, é calculada a distância do centro do *robot* até um dos seus vértices, multiplicando-a por dois, com vista a obter a distância a que o nosso *robot* tem de passar do centro dos obstáculos. Este valor teve de ser arredondado para cima em três casas decimais para remover *bugs* de colisões por arredondamento de variáveis.

Após saber a distância e o ângulo necessários para contornar os obstáculos, resta calcular o ponto correspondente. Mais uma vez, como o *heading* dos *RockQuads* não influencia as suas *hitboxes*, sabe-se que os ângulos dos vértices são sempre  $45^\circ$ ,  $135^\circ$ ,  $225^\circ$  e  $315^\circ$ .

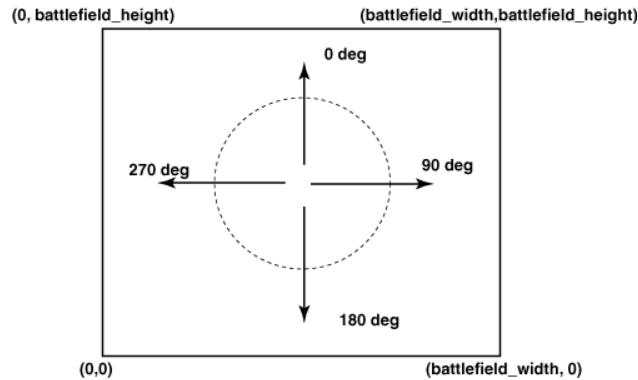


Figura 3: Ângulos e Coordenadas do RoboCode

Tendo em conta a nossa implementação, os ângulos necessários serão os dos vértices a  $45^\circ$ ,  $135^\circ$  e  $315^\circ$ , pois, como vai ser possível de ser analisado nas figuras dos casos genéricos, como na figura 4 e dos atalhos, como na figura 6, o ângulo  $225^\circ$  nunca irá diminuir a distância percorrida e, portanto, não será utilizado.

Os vértices relevantes de cada obstáculo dependem da posição anterior do nosso *robot*, das coordenadas do próprio obstáculo e das do próximo obstáculo. Assim, nestes casos genéricos, o nosso *robot* vai sempre passar pelo vértice de ângulo  $315^\circ$  do primeiro obstáculo e, depois, dependendo se o próximo obstáculo tem o valor da coordenada  $y$  mais elevado ou não, o próximo ponto é escolhido no primeiro ou segundo obstáculos. O vértice de ângulo  $45^\circ$  do segundo obstáculo será também sempre utilizado, bem como vértice de ângulo  $135^\circ$  do terceiro. O ponto intermédio entre estes depende do valor da coordenada  $x$  do segundo e terceiro obstáculos. Estes casos podem ser visualizados na figura 4.

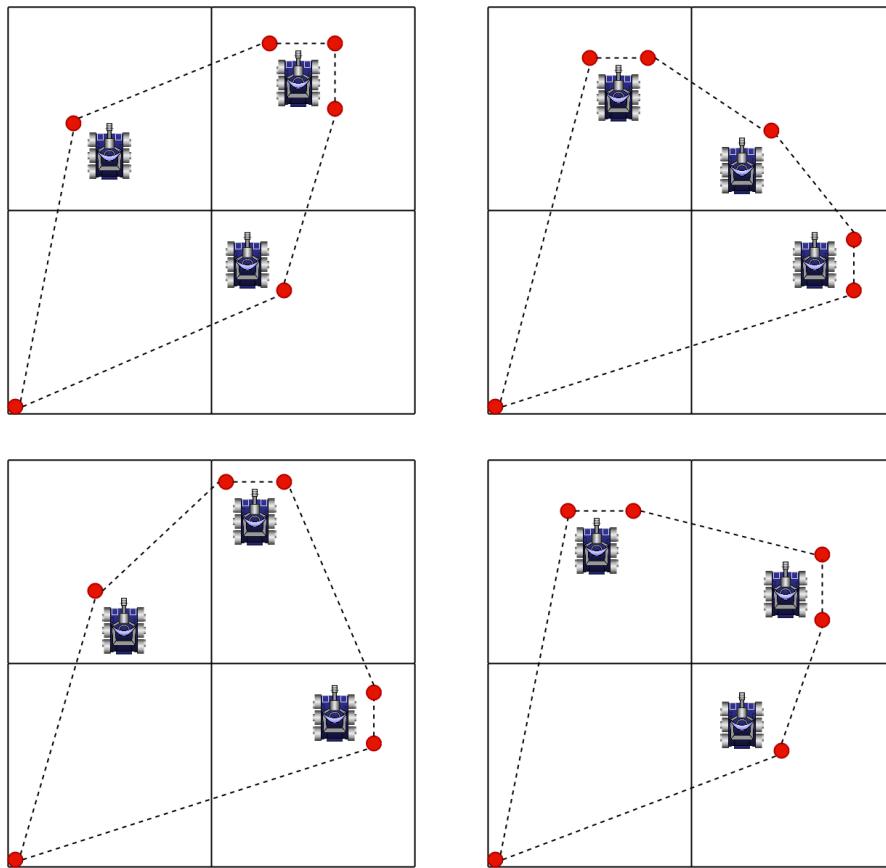


Figura 4: Casos mais genéricos de circum-navegação

Com os *RockQuads* fornecidos pelos docentes é difícil acontecer disposições “extremas”, isto é, uma disposição onde pode ser feito um atalho entre o primeiro e terceiro obstáculos (como se pode ver na figura 5), todavia a nossa implementação tem esses casos em conta. Estes casos referidos podem levar a que a eficiência ultrapasse os 100%, uma vez que o perímetro entre os obstáculos com a origem pode ultrapassar a distância necessária a percorrer para os contornar. Estes casos extremos acontecem quando um dos obstáculos pode ser ignorado.

A abordagem tomada para encontrar estes “atalhos” realiza-se através do cálculo de dois ângulos - um entre a antepenúltima e a última posição calculadas e um dos dois próximos potenciais pontos e outro entre essas mesmas posições e o outro dos dois próximos poten-

ciais pontos (um em cada obstáculo). Desta forma, o próximo ponto é escolhido tendo em conta o ângulo maior, podendo, assim, passar um obstáculo à frente.

Na figura 5 é possível visualizar o percurso onde a eficiência ultrapassa os 100%, assim como os ângulos calculados na decisão de efetuar o atalho.

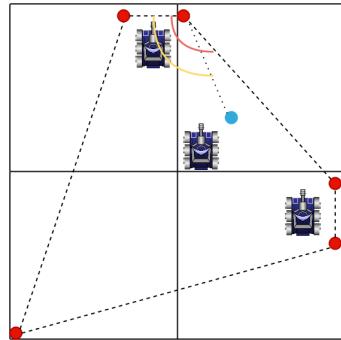


Figura 5: Percurso com a melhor eficiência e os ângulos para os atalhos

Os três atalhos possíveis de ser realizados podem ser visualizados na figura seguinte.

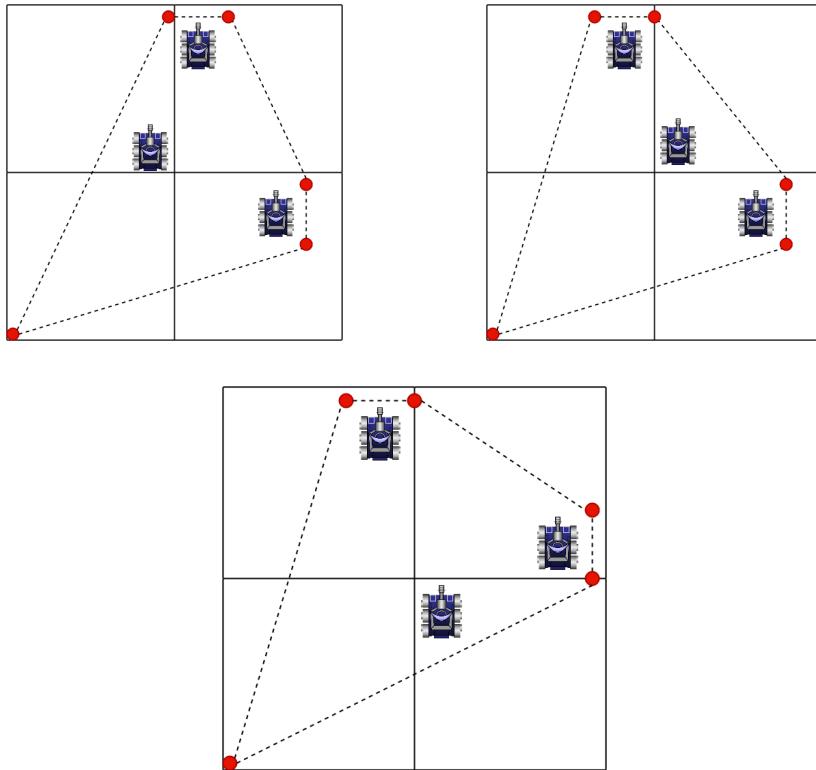


Figura 6: Casos de atalhos na circum-navegação

### 3 Fase 2

A segunda fase tinha como objetivo a implementação de vários comportamentos em equipa, ocorrendo a cooperação entre agentes, isto é, entre os *robots* da equipa, de forma a apresentar comportamentos de grupo ou sociais. Desta forma, procedeu-se ao desenvolvimento de várias equipas, com vista a exemplificar os vários comportamentos diferentes.

É de ressalvar que o *robot Grounded* (a amarelo) apenas é colocado em jogo devido às limitações do Robocode, pois têm de existir, pelo menos, duas equipas no campo de batalha para esta poder ocorrer.

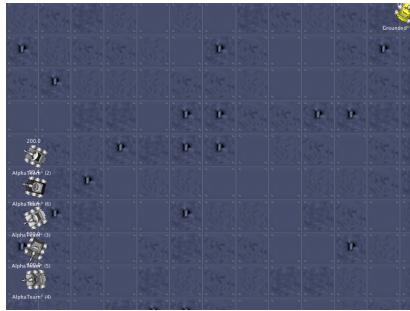
#### 3.1 Primeira versão da *March* (Fila de robots)

Nesta equipa, o objetivo passa por constituir um comportamento em marcha, composto por cinco elementos, sendo um deles o coordenador.

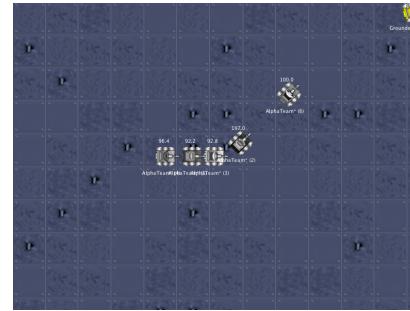
Inicialmente, o *robot* coordenador calcula todos os pontos por onde a marcha irá passar, sendo também definido o ponto inicial para toda a equipa. Após todos os *robots* estarem na sua posição inicial, ficam a aguardar pela próxima ordem do seu coordenador. Este, quando avança no terreno de batalha e após chegar ao ponto seguinte da rota, envia uma mensagem a todos os seus companheiros de equipa com a posição para a qual se devem mover. Posto isto, permanece a aguardar que os restantes elementos de equipa confirmem a sua chegada à posição indicada para se poder mover para a próxima posição.

Aqui realça-se o comportamento uniforme entre os seguidores do coordenador, visto que todos avançam praticamente ao mesmo tempo, de forma sincronizada.

Tanto o coordenador como os restantes elementos possuem uma *Condition* com o objetivo de confirmar se podem dar o próximo passo. No caso do coordenador, a *Condition* passa a *true* quando os restantes elementos de equipa indicam que estão prontos. Para os restantes elementos de equipa, estes apenas aguardam pela mensagem do coordenador.



(a) Início da formação da AlphaTeam



(b) Movimentação síncrona da AlphaTeam

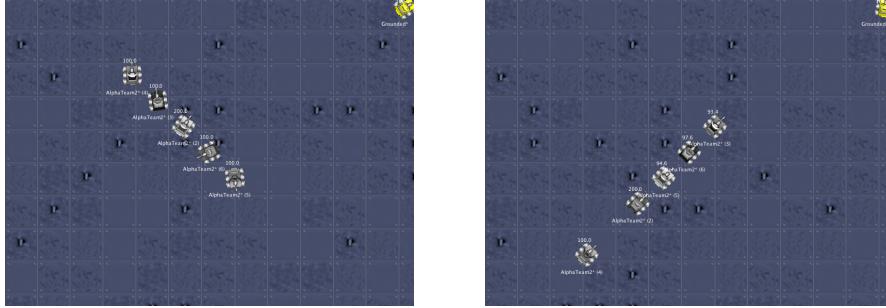
Figura 7: Formação em *March 1*

#### 3.2 Segunda versão da *March* (Fila de robots)

Esta segunda versão da *March* tem o comportamento similar, visto que todos os *robots* se situam numa dada posição inicial e percorrem um circuito armazenado num *ArrayList*. No entanto, a comunicação entre eles sofreu alterações, sendo que cada *robot* apenas comunica com o *robot* que se situa exatamente na posição anterior, à exceção do último que comunica com o líder (primeiro). Ora, isto provoca um movimento em cadeia, pois um *robot* só se pode movimentar para a posição seguinte após receber a mensagem (a informar que a posição para onde este quer se deslocar já não está ocupada) do seu antecessor. Para

tornar este comportamento possível, todos os *robots* sabem o trajeto na sua completude, contrariamente à primeira marcha.

Desta vez, a Condition utilizada é similar para todos os elementos da equipa, que apenas necessitam de uma mensagem de confirmação de outro elemento para passar para *true*.



(a) Início da formação da AlphaTeam2

(b) Movimentação assíncrona da AlphaTeam2

Figura 8: Formação em *March 2*

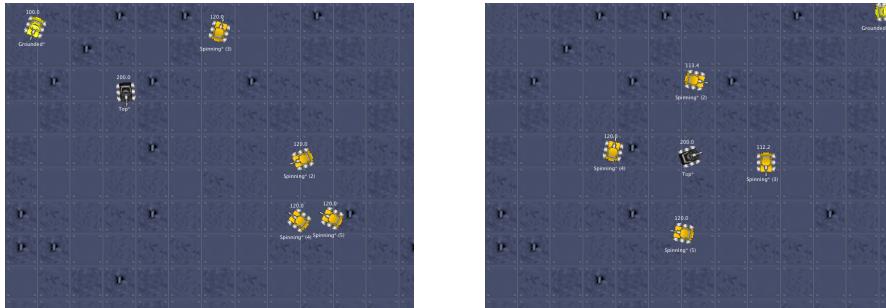
### 3.3 Circular em “rodinha”

O objetivo desta equipa é ter quatro *robots* a circular em redor do *robot* capitão, acompanhando o seu movimento e mantendo sempre a forma de uma circunferência. Para isso, foram criadas duas classes de *robots* - o *robot* central e os “rodeadores”.

Essencialmente, o *robot* central atribui um *id* a cada um dos restantes colegas e envia-lhes uma posição inicial, um ângulo e um raio. A partir deste momento, apenas envia o seu deslocamento e os restantes *robots* calculam a posição do colega central (ou seja, o centro da circunferência formada). Com este ponto, incrementam o ângulo, calculando e deslocando-se para a sua nova posição (relativa ao centro da circunferência referida). O movimento do *robot* central é um percurso pré-definido, efetuado 125 vezes.

Para manter a sincronia, é necessário existir comunicação e utilizar várias Conditions que permitem que os *robots* aguardem pela receção de mensagens importantes como, por exemplo, receber a posição inicial (sendo enviada uma mensagem ao líder quando a atingirem) e a ordem para começar o percurso.

Visto que a comunicação entre dois *robots* é bastante importante nesta implementação, bem como não sobrecarregar os *robots*, optou-se por implementar uma função que permite obter o nome de todos os colegas de equipa e, desta forma, conseguir enviar mensagens para um colega específico.



(a) Fase Inicial

(b) Fase Final

Figura 9: Formação em Roda

### 3.4 Formação em V

O objetivo desta equipa é ter cinco *robots* dispostos em V, a imitar/acompanhar o *robot* central, mantendo sempre a forma de um V. Para isso, foram criadas duas classes de *robots* - o *robot* líder e os laterais. Como não são usados radares nesta equipa, todos os *robots* são *droids*.

Essencialmente, o *robot* central atribui um *id* a cada um dos restantes colegas e envia-lhes uma posição inicial. A partir deste momento, envia o seu deslocamento e os restantes *robots* apenas necessitam de se deslocar na mesma direção e distância. Os *robots* têm um percurso pré-definido e perfazem-no 200 vezes.

Para manter a sincronia, é necessário existir comunicação e utilizar várias *Conditions* que permitem que os *robots* aguardem pela receção de mensagens importantes como, por exemplo, receber a posição inicial.

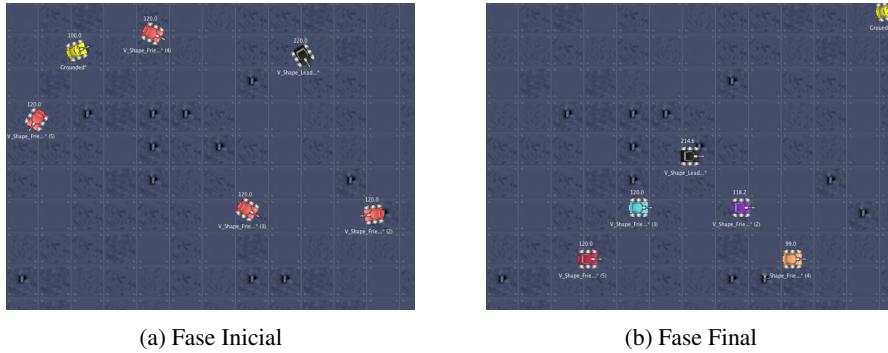


Figura 10: Formação em V

### 3.5 Hitman

Já a pensar na fase 3, fase esta que tem como objetivo construir uma equipa capaz de vencer todas as batalhas, implementou-se um comportamento social que consiste em ter um *robot* coordenador, denominado por *ScannerLeader*, que, numa fase inicial, se desloca para uma dada posição, realizando imediatamente o *scan* de todos os seus colegas (*SpinnerDroids*), sendo que estes *droids* são posteriormente ordenados para uma posição estratégica. Após receberem essa informação, estes deslocam-se para a posição referida. A partir deste momento, entra-se na fase final, onde o coordenador passa a detetar os inimigos que estão no campo de batalha e, quando estes são detetados, todos os *droids* são informados sobre qual o inimigo a abater, disparando contra ele.

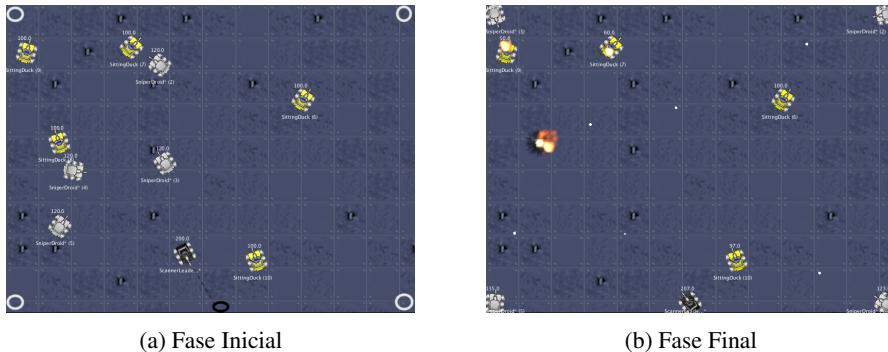


Figura 11: Formação Hitman

### 3.5.1 Processo de comunicação

De seguida, apresenta-se a troca de mensagens para realizar o comportamento detalhado anteriormente:

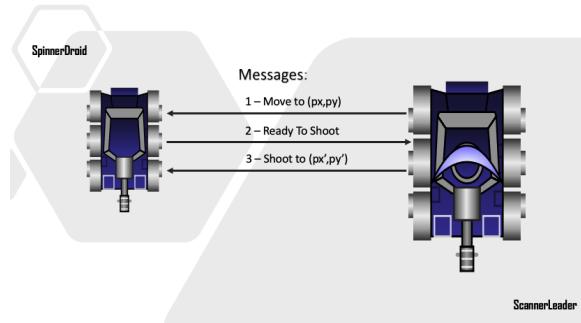


Figura 12: Comunicação entre o líder e um *droid*

1. O *ScannerLeader*, na fase inicial, faz o *scan* dos companheiros de equipa, atribuindo uma posição num dos cantos do mapa ao primeiro colega detetado. De seguida, repete o processo para os restantes colegas de equipa, ordenando que cada um destes se desloque para um canto do mapa ainda não atribuído;
2. O *SpinnerDroid*, aquando da receção da mensagem, desloca-se para as coordenadas recebidas. Quando chegar a esse ponto, envia uma mensagem ao líder a informar que está pronto para atacar;
3. O *ScannerLeader* aguarda que todos os *SpinnerDroids* lhe enviem a confirmação referida. Após essa confirmação, começa a tentar detetar inimigos e, sempre que encontra um, calcula a sua posição e ordena todos os *robots* a disparar para essas coordenadas.

### 3.6 Tag (“caçadinhas”)

O objetivo desta formação passa por tentar recriar o popular jogo “caçadinhas”. Aqui é escolhido um *robot* para começar a perseguir os colegas, informando-os de que o vai fazer. Quando atingir outro *robot*, passam ambos a perseguir e o novo perseguidor informa os restantes que mudou de papel.

No que toca às estratégias, os *robots* perseguidores deslocam-se na direção dos restantes *robots*, enquanto que os outros se deslocam na direção oposta, tentando evadir. Para tornar o jogo mais simples de acompanhar, os *robots* têm cores diferentes consoante o seu papel.

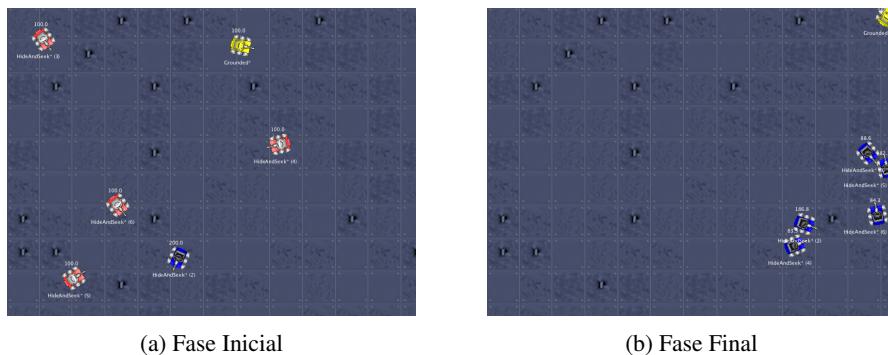


Figura 13: Formação *Tag*

Na figura 13a, conseguimos visualizar a fase inicial do jogo, onde apenas um *robot* está a apanhar (a azul). Este jogo das “caçadinhas” só termina quando **todos** os *robots* forem apanhados, como se pode ver na figura 13b.

### 3.7 Ratchet & Clank

Esta equipa é composta por apenas três elementos: dois *Ratchets*, que vão simular as raquetes e um *Clank*, que vai ter o papel da bola.

Inicialmente, os *Ratchets* dirigem-se para as laterais do mapa e o *Clank* para o centro. Assim que cada *Ratchet* chega à sua posição, envia uma mensagem ao *Clank* a informar que já se encontra pronto para começar a partida. Após receber as duas mensagens (de cada *Ratchet*), o *Clank* inicializa a sua marcha para a frente. Sempre que este bate numa parede faz “ricochete”, ou seja, o ângulo que faz com a parede quando embate contra ela é o mesmo que faz quando sai ricocheteado.

É o papel dos *Ratchets* tentar prever onde o *Clank* vai estar ao cruzar a sua linha de defesa. Para este efeito, é efetuado o *scan* reiteradamente, de forma a ter atualizada a informação útil sobre o *Clank* para poder efetuar os cálculos com a sua posição, velocidade, direção e sentido. São tidos em conta os possíveis ricochetes para a previsão de onde estará o *Clank* quando intersetar a linha de defesa.

Aquando de uma defesa por parte de um *Ratchet*, ou seja, quando este embate no *Clank*, envia-lhe uma mensagem com a nova direção que quer que o *Clank* tome. Quando há uma falha na defesa, isto é, o *Clank* passa uma das linhas de defesa dos *Ratchets*, este envia uma mensagem aos dois jogadores a informar que a jogada acabou e dirigem-se todos para o centro do campo.

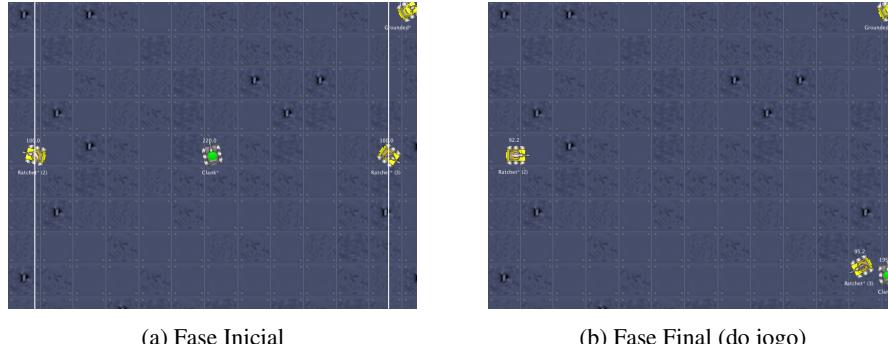


Figura 14: Formação *pong* do *Ratchet & Clank*

## 4 Fase 3

### 4.1 Funções relevantes

Para esta fase foram criadas algumas funções necessárias para obter um rendimento maior no campo de batalha e “criar” decisões mais complexas, estando estas explicadas nas secções 4.1.1 a 4.1.3.

#### 4.1.1 Previsão da posição futura do inimigo

Para permitir obter resultados melhores no que diz respeito às batalhas, optou-se por implementar um algoritmo que permite calcular uma aproximação da posição futura dos *robots já scaneados*, para o movimento linear. Para isso, é necessário armazenar várias informações sobre o *robot* inimigo como, por exemplo, a sua posição, velocidade e ângulo, bem como a velocidade da bala e o instante em que o *robot* foi avistado pela última vez.

Primeiramente, é necessário calcular a distância entre o nosso *robot* e o *robot* inimigo. Utilizando esta distância, é possível calcular o tempo necessário para a bala a percorrer.

$$v_b = \frac{d}{t_b} \equiv t_b = \frac{d}{v_b}$$

No entanto, este tempo não é o mais correto, pois não tem em conta o tempo necessário para a bala se deslocar até à posição futura. Assim, decidimos fazer uma aproximação a essa posição<sup>3</sup>.

$$p' = p + v \cdot t = (x, y) + (v \cdot \cos(\alpha), v \cdot \sin(\alpha)) \cdot t = (x + v \cdot \cos(\alpha) \cdot t, y + v \cdot \sin(\alpha) \cdot t)$$

A partir desta posição aproximada, calcula-se uma nova distância aproximada e, de seguida, o tempo necessário para a bala viajar até essa posição.

Este cálculo, ainda assim, continha alguns erros de aproximação e, para os mitigar, optou-se por efetuar três aproximações, permitindo uma maior convergência do valor.

No final, já com o tempo calculado, calcula-se, então, o ponto futuro a partir da fórmula acima.

Após a primeira competição intraturma, decidimos que ainda existiam alguns aspetos a melhorar:

- a previsão da posição explicada acima passou a ter em conta a aceleração e velocidade angular, permitindo “acompanhar” um movimento não retílineo ou não constante;
- efetuou-se uma aproximação por cada turno “passado”, através da criação um ciclo para recrutar o percurso de um inimigo em todos os turnos até a bala o atingir, permitindo a melhoria referida no ponto prévio;
- considerou-se a possibilidade de o inimigo embater contra as paredes do mapa, o que faz com que o seu movimento cesse.

#### 4.1.2 Mitigação dos disparos entre colegas de equipa (*friendly-fire*)

Depois de vários testes, constatou-se que um dos problemas existentes nas nossas equipas era o *friendly-fire*. Para o tentar mitigar, utilizou-se a técnica de os *robots* não dispararem

<sup>3</sup> No cálculo da posição futura será considerada a diferença entre o momento em que ela é calculada e o instante em que a informação sobre o estado do *robot* é recolhida.

em direção aos colegas.

Assim sendo, é necessário que cada *robot* verifique se um colega (através das suas posições atuais) se encontra no alinhamento do disparo para o inimigo, através do cálculo de um *threshold*, mais precisamente:

$$\tan^{-1} \left( \frac{26}{d - 52} \right)$$

O valor 26 (e o 52, por ser o seu dobro) foi escolhido, pois é ligeiramente superior a  $\sqrt{18^2 + 18^2} \approx 25,5$ , que corresponde a metade do comprimento máximo de cada *robot*. Assim, caso o ângulo para o inimigo se encontre dentro do *threshold* referido (em relação ao centro de cada colega), o *robot* decide não disparar.

#### 4.1.3 Mitigação dos embates entre colegas de equipa (*friendly-hit*)

Tal como com o *friendly-fire*, constatou-se que um dos problemas que impedia a equipa de se tornar mais eficiente era o facto de existir bastantes embates entre os *robots* da nossa equipa. Assim, decidiu-se implementar uma função que permite perseguir os inimigos, tendo em conta as posições dos colegas de equipa. Desta forma, sempre que é calculada uma posição para onde se deve deslocar, o *robot* verifica primeiro se irá embater em algum colega. Se este for o caso, o *robot* tenta encontrar um caminho alternativo, contornando os colegas.

## 4.2 A Nossa Equipa

A equipa implementada foi alvo de elevada experimentação, isto é, foram efetuadas várias batalhas contra as mais variadas equipas (referidas na secção 5) encontradas em páginas oficiais de competições do RoboCode. As modificações aplicadas à nossa equipa foram sempre realizadas tendo em atenção a prestação geral contra essas equipas referidas. Todavia, algumas das implementações acabaram por não mostrar vantagens em batalha, como é o caso de tentar evitar o *friendly-fire*.

Nas próximas secções serão explicados os pontos mais importantes sobre a implementação da nossa equipa.

### 4.2.1 Distribuição de IDs

Para a distribuição de *ids* é necessário, em primeiro lugar, que os elementos de equipa permitam conhecer-se uns aos outros, visto que, no nosso entender, a função `isTeammate()` do RoboCode não se encontra a funcionar corretamente (esta devolve sempre `false` independentemente do *scanned robot*).

Desta forma, a primeira ação que os elementos da equipa fazem é enviar uma mensagem em *broadcast* para os restantes elementos, informando-os do seu nome, sendo que toda a equipa guarda os nomes dos companheiros. Assim, é possível saber a qualquer momento se um *scanned robot* faz ou não parte da equipa.

Posto isto, ao termos cada elemento da equipa com acesso aos nomes dos restantes, consegue-se, de modo universal, atribuir um *id* a cada elemento, através da comparação entre *strings* (isto é, cada elemento compara o seu próprio nome com os restantes elementos da equipa).

```
for (String s : this.getTeam())
    if (myName.compareTo(s) > 0) id++;
```

#### 4.2.2 Scanning e Movimentação

O radar é dos elementos mais importantes para a nossa equipa, pois é aquando do *scan* de um inimigo que a maior parte do código é executado, isto é, a movimentação (perseguindo o inimigo), o virar da arma e do radar e a decisão de disparo.

Inicialmente, no *scan* do inimigo são efetuados os cálculos e um *set* do radar para o centro do inimigo, com vista a mantê-lo na “mira”. De seguida, é calculado o ponto para onde o nosso *robot* tem de se mover, consoante a estratégia, sendo que o movimento para tal ponto ocorre recorrendo a várias ações *set*. É, também, executado o virar da arma para o inimigo, através da previsão da sua posição futura e, por fim, é disparada uma bala com o *power* calculado através de uma função que tem a distância ao inimigo como variável. Tanto o virar como o disparar da arma não são *sets*, o que faz com que o disparo ocorra logo após à arma ser virada.

Para a movimentação são apenas utilizados *sets*, tanto para andar para frente e para trás, como para virar. Deste modo, o *robot* passa o mínimo de tempo parado e cria um movimento menos previsível. As ações mais rápidas para chegar ao destino são sempre executadas, andando para trás quando necessário. Por exemplo, se o ponto alvo se encontra a 175° do nosso *robot*, este apenas vai virar 5° para a esquerda e andar de marcha-atrás, ao invés de virar 175° para a direita e andar em frente.

Sempre que o inimigo escapa de “vista” ou é destruído, o radar é rodado infinitamente até o mesmo ou um novo inimigo ser encontrado.

#### 4.2.3 Estratégia e ordem de ataque

Inicialmente, a estratégia delineada foi o ataque em grupo, ou seja, todos os *robots* atacam o mesmo inimigo, contudo verificou-se que dividir a equipa em dois trazia melhores resultados. Desta forma, o líder é quem decide a ordem de ataque criando duas listas, sendo uma percorrida por completo antes de percorrer a outra. Esta divisão é feita para serem atacados primeiro os *robots* não *droids*, pertencentes à primeira lista e, posteriormente, os *robots droids*, pertencentes à segunda lista.

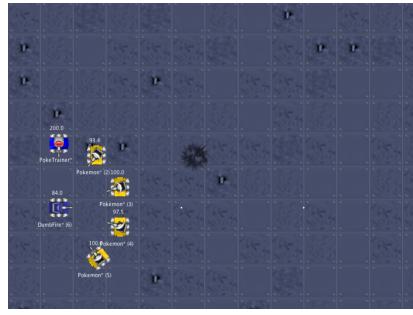
A energia inicial de cada *robot* inimigo é utilizada para definir que tipo de *robot* é - se tiver 200 de energia, é um líder normal, já para ser um líder *droid*, necessita de 220; caso disponha de 100 de energia é um *robot* normal e, se apresentar 120 de energia, é um *droid* normal.

Após alguns testes, chegou-se à conclusão que atacar primeiro o líder inimigo traz alguma vantagem no geral e, por essa razão, esse *robot* é colocado na lista como o primeiro a ser atacado.

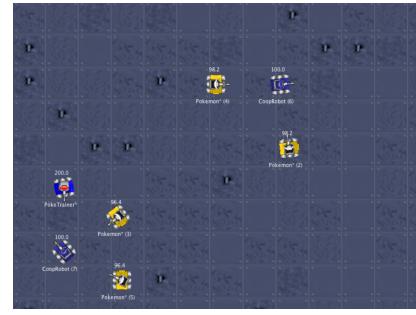
Como referido anteriormente, a equipa divide-se em dois conjuntos, sendo que os elementos de cada conjunto atacam o mesmo inimigo. Uma vez que as equipas são formadas por cinco elementos, um conjunto tem três e outro dois elementos. Esta distribuição é realizada através do *id* de cada elemento da equipa, sendo que os *ids* pares formam o de três elementos (0, 2 e 4) e os *ids* ímpares integram o de dois elementos (1 e 3). O conjunto dos pares percorre as listas da esquerda para a direita e os ímpares percorrem as listas da direita para a esquerda, de modo que os cinco elementos se encontrem, o mínimo de vezes possível, a atacar o mesmo inimigo.

Como podemos observar na figura 15, é utilizada uma formação entre um quarto e pouco menos de meia circunferência para “circundar” o inimigo (em relação ao centro do campo

de batalha), sendo que isso permite evitar o *friendly-fire* e os *friendly-hits*. A vantagem numérica desta formação proporciona rapidez no ataque ao inimigo, auxiliando a sobrevivência geral da nossa equipa.



(a) Formação com inimigo único



(b) Formação com vários inimigos

Figura 15: Diferentes estados da formação

#### 4.2.4 *OnHitRobot* e *OnRobotDeath*

Para permitir que a equipa se possa defender, decidiu-se incorporar o comportamento de resposta a um embate. Quando se trata de um colega de equipa, o *robot* apenas tenta sair do caminho dele, ao passo que, quando se trata de um inimigo, o *robot* riposta com um disparo, cuja potência é definida pelo ângulo necessário para o atingir.

Se um inimigo que está a ser perseguido for abatido, é escolhido um novo inimigo, tendo em conta a ordem de ataque já referida.

## 5 Análise de Resultados

Para a fase 1, foi testada a circum-navegação com vários valores de distanciamento aos *robots* obstáculo, tendo obtido o melhor resultado (sem aumentar a taxa de colisão) com distanciamento igual a 50.912, aproximadamente igual ao resultado de  $\sqrt{18^2 + 18^2} * 2$ .

Já para a fase 3, realizámos várias batalhas contra os *robots* presentes no *TeamRumble*<sup>4</sup>, mais precisamente, contra as equipas CombatTeam, Polylunar e OmegaTeam.

Para as últimas duas equipas, obtivemos resultados bastante satisfatórios, vencendo cerca de 99% e 90% das batalhas e 78% e 70% dos pontos, respetivamente, todavia, contra a primeira, apenas alcançámos cerca de 1% de vitórias e 20% dos pontos.

Após a vitória no primeiro torneio intraturma, optámos por manter a mesma estratégia, tentando melhorar o posicionamento em torno dos inimigos e a previsão das suas posições.

### 5.1 Trabalho futuro

Após várias batalhas de teste efetuadas contra as equipas referidas, percebemos que bastantes melhorias desenvolvidas não surtiam o efeito desejado no desfecho das batalhas e, muitas vezes, diminuíam a percentagem de vitórias e pontos ganhos durante a batalha.

Para além das melhorias referidas na secção 4 foi, ainda, implementada uma função para criar oscilação no movimento dos *robots* para que se tornassem alvos mais imprevisíveis, fazendo com que fossem menos atingidos pelas balas inimigas, contudo, à imagem das duas funções de mitigação (*friendly-fire* e *friendly-hit*), a *performance* da equipa baixava. Assim sendo, optámos por utilizar versões mais simples e menos “deliberativas”, pois a simplicidade da solução obtida permite resultados mais abonatórios.

Posto isto, numa possível continuação futura, o nosso maior foco seria a melhoria das funções de mitigação, guardando as posições mais recentes dos inimigos, para tentar criar uma estimativa da sua posição futura, através de padrões das ações passadas.

Por fim, também poderia ser investigada/desenvolvida uma função de movimento mais “certeira”, sem perder a capacidade de movimento rápido, mas com alguma incerteza, para garantir que não se deslocava num movimento retilíneo ou circular.

### 5.2 Dificuldades

Durante o desenvolvimento deste projeto, a necessidade de criar funções para substituir as funções da classe TeamRobot (como *isTeammate*) foi um entrave, pois obriga a nossa equipa a “gastar” tempo a comunicar o seu nome aos restantes colegas. Para além disso, o facto de certas operações serem executadas em paralelo (como *setTurn*) e certas ações interromperem outras (por exemplo, o *onHitRobot* interrompe um *turn*), tornou o desenvolvimento mais demorado, pois obrigou-nos a perceber/mitigar esta execução, visto que certas instruções (por vezes, contraditórias) eram executadas concorrentemente ou interrompiam outras.

Inicialmente, a nossa estratégia passava por cada *robot* ser responsável por *scanear* um inimigo, tentando sempre ter noção da posição dele no mapa, partilhando-a com os colegas. Todavia, esta abordagem tinha vários problemas, como perseguir um inimigo apenas com o radar, acabando por o perder várias vezes, e a má previsão da posição dos inimigos. Este último problema foi o grande responsável por abandonarmos esta abordagem, pois, como a informação sobre o inimigo estava muito desatualizada, a posição futura era prevista através de informação recebida dos colegas, o que fazia com que os disparos errassem muitas vezes.

<sup>4</sup> <http://robocode-archive.strangeautomata.com/robots/>

## 6 Conclusão

Durante a realização deste projeto, tivemos a oportunidade de consolidar e desenvolver os conhecimentos abordados nesta unidade curricular, bem como na unidade curricular de Agentes Inteligentes, nomeadamente, comunicação, comportamentos sociais, estratégias de movimentação de *robots*, técnicas de previsão (de tiro e posição futura), sonorização do ambiente, através do *scanner* ou sensores de toque, entre outros.

Para pôr estes conceitos em prática criámos *robots* e equipas de *robots* (para combater) no jogo RoboCode, um ambiente de programação em Java que tem como objetivo a construção de *robots*, sendo também permitida a simulação e visualização deste combate. Foram, ainda, desenvolvidos os mecanismos de deliberação dos *robots* referidos.

Assim, é aberta a possibilidade de exploração de diversas áreas da inteligência artificial.

Como já referido, este projeto é constituído por três etapas:

- a primeira etapa é uma fase introdutória do jogo, pois deve ser implementada, de forma eficiente, uma possível circum-navegação de obstáculos, circum-navegação esta que deve ser registada pelo odómetro criado, para calcular a distância percorrida pelo *robot* durante o percurso. Assim, considerámos que obtivemos uma eficiência bastante satisfatória no contorno dos obstáculos, tendo em conta a problemática;
- na segunda fase foram criados vários comportamentos de grupo que demonstram a cooperação entre os agentes (*robots*), como movimentação em grupo (pela equipa AlphaTeam), comunicação em grupo (pela equipa Tag) e execução de tarefas delineadas por um líder (pela equipa Hitman). Assim, considerámos que desenvolvemos comportamentos complexos que demonstram toda a flexibilidade do RoboCode;
- na terceira fase, cujo principal objetivo é derrotar uma equipa inimiga num cenário de batalha, foi desenvolvida uma equipa para o cumprir, possuindo como principais características o seu posicionamento reativo (tornando a sua movimentação dependente do inimigo, minimizando o tempo que permanece numa dada posição), a capacidade de divisão do grupo, a previsão da posição de um inimigo (de forma a prever a posição para o disparo), deliberação da potência do disparo, entre outras.

Para além do trabalho feito para as três fases, foram, ainda, testadas várias possíveis melhorias, o que permitiu aprofundar o nosso conhecimento das físicas e dinâmicas de jogo, assim como perceber que tipos de abordagem “colhiam mais frutos”. Apesar das dificuldades enumeradas, sentimos que a equipa da fase 3 é capaz de fazer frente a equipas que utilizem o mesmo esforço computacional que a nossa.

Em suma, consideramos que o trabalho produzido cumpre todos os requisitos propostos, sendo que, na fase final, apresentámos uma equipa com boa performance contra *robots* de alta competição, apresentando boas classificações, tanto a nível pontual, como a nível de vitórias, uma vez que fomos capazes de vencer contra várias equipas.