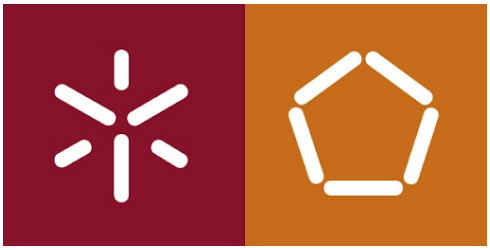

Universidade do Minho

Mestrado em Engenharia Informática

Conceção e implementação de um sistema Multiagente

Agente e Sistemas Multiagente

Grupo 09

Gonçalo Almeida (pg47212)
Leonardo Marreiros (pg47398)
Maria Sofia Marques (pg47489)
Pedro Fernandes (pg47559)

24 de setembro de 2022

Conteúdo

Lista de Figuras **ii**

1 Introdução **1**

2 Estado de Arte **1**

2.1 Descrição do Problema 1

3 Contextualização da Solução **2**

4 Arquitetura **3**

5 Protocolo de Comunicação **5**

5.1 Iniciar Arbitro, Líder e Jogadores 5

5.2 Jogador Informa Estado Atual 6

5.3 Árbitro Informa Estado Atual 7

5.4 Estratégia Definida pelo Líder 8

5.5 Diagrama de Comunicação 9

6 Processamento Interno dos Agentes **10**

6.1 Agente Jogador 10

6.2 Agente Líder 11

6.3 Agente Árbitro 11

7 Implementação dos Agentes **11**

7.1 Árbitro 12

7.2 Líder 12

7.3 Jogador 13

8 Avaliação e testes de performance dos agentes **14**

8.1 Ferramenta de visualização 14

8.2 Funções Objetivo e Análise de Estratégias 15

9 Conclusão e Trabalho Futuro **17**

10 Anexos **19**

10.1 Código de cálculo do caminho mais curto 19

10.2 Demonstração do código acima 22

Lista de Figuras

1 Diagrama de Classe 4
2 Diagrama de sequencia: inicio do jogo 6
3 Diagrama de sequencia: *Update* posições do Jogador 7
4 Diagrama de sequencia: Morte de Jogador 8
5 Diagrama de sequencia: Estratégia de jogo 9
6 Diagrama de comunicação 10
7 Diagrama de Atividades 10
8 Textura 1 de areia 14
9 Textura 2 de areia 14
10 Jogador da equipa azul 14
11 Jogador da equipa vermelha 14
12 Mapa de jogo 15
13 Resultados jogos entre equipas totalmente ofensivas e totalmente defensivas 16
14 Resultados jogos entre equipas totalmente ofensivas e equipa mista 17

1 Introdução

Sistemas Multiagentes são uma sub área da Inteligência Artificial Distribuída e concentram-se no estudo de agentes autónomos no universo multiagente. Neste tipo de sistemas, um número limitado de agentes torna-se capaz de sensorizar e interpretar a informação adquirida de um respetivo ambiente virtual, e consequentemente, tomar decisões inteligentes, de forma cooperativa e/ou competitiva com agentes vizinhos, sendo estas decisões baseadas num objetivo concreto.

O objetivo do seguinte sistema multiagente consiste na elaboração de um conjunto de agentes inteligentes autónomos que irão explorar um mundo bi-dimensional (i.e., coordenadas x,y, com dimensão 35x35), onde serão agrupados em duas equipas (equipa vermelha e azul), composta por 5 agentes. Cada um destes agentes tem o objetivo de "explorar o mundo"com o intuito de descobrir a localização dos agentes oponentes e colaborar com os seus aliados para encontrar a melhor forma de perseguir e eliminar todos os agentes da equipa adversária (inimigos).

Posto isto, no seguinte documento será apresentado um estado de arte sobre os agentes, as sua respetivas definições e aplicações no sistema, será explicado como é que foi pensado, concebido e modelado uma arquitetura distribuída com base nos agentes para o respetivo problema. Para tal, será apresentada uma contextualização geral da nossa solução, a respetiva arquitetura e, por fim, o respetivo protocolo de comunicação. Nestes três últimos tópicos foram aplicadas metodologias *Agente UML* com o intuito de formalizar os protocolos de interação entre os agentes. Para finalizar, será apresentada uma breve descrição da parte gráfica deste projeto.

Após feita a introdução conceptual e modelação ao sistema é descrita a implementação feita com o recurso JADE para a criação, manutenção e ação dos agentes.

2 Estado de Arte

2.1 Descrição do Problema

Tal como dito anteriormente, este sistema multiagente consiste num simulador de estratégia de combate entre duas equipas oponentes. Para a concretização deste projeto foram utilizados 3 tipos de agentes: o **agente participativo** (Jogador), representam a equipa e são estes que vão circular pelo mapa; **o agente central** (Líder), um presente em cada equipa, tem como o objetivo coordenar os seus aliados da forma mais eficiente possível. Estas decisões são influenciadas por um conjunto de fatores existentes no meio, tais como quem tem mais inimigos no seu campo de visão e/ou o inimigo mais perto do seu jogador; Por fim, um **agente secundário** (Árbitro) que verifica todo o tipo de acontecimentos e reporta ao líder da equipa correspondente. Por exemplo, caso um jogador tenha sido cercado, é o árbitro que indica ao Líder dessa equipa que aquele jogador foi eliminado.

O principal objetivo deste simulador passa por monitorizar e eliminar a equipa contrária

a partir da informação que o Líder de cada equipa tem ao seu dispor, o mais rapidamente possível, recorrendo a estratégias que aumentem a eficácia para tal, o jogo termina quando restar jogadores apenas de uma equipa. Quanto às regras do jogo, um agente é apenas considerado eliminado se este for cercado pela equipa oponente nas quatros direções (uma ou duas destas direções podem ser as paredes), cada agente tem um campo de visão de 7x7 (só consegue ver oponentes que estejam, no máximo a 3 $u.m$ de distância em qualquer direção) e, por fim, agentes só podem comunicar com agentes da sua mesma equipa.

3 Contextualização da Solução

No inicio de cada simulação será construído um mapa bi-dimensional constituído por 35 $u.m$ de comprimento (eixo x) e 35 $u.m$ de altura (eixo y). Os agentes participativos (Jogadores) tanto podem ser colocados aleatoriamente ou de forma estratégica dentro do mapa, os agentes central e o agente secundário encontram-se fora do mapa (uma vez que não se mexem não são necessários). Assim que começa o jogo, os jogadores recebem um tipo (defensivo ou ofensivo) e respeitam uma hierarquia de perigo. No caso em que diferentes aliados tenham diferentes inimigos no seu campo de visão, o Líder escolhe um "jogador central", este será o jogador que o resto dos aliados irão socorrer. Uma vez que é um jogo de estratégia/sobrevivência foi decidido que este jogador central será quem tiver mais inimigos no seu campo de visão, em caso de empate é escolhido o que tiver mais aliados por perto.

Os agentes participativos vão ter três tipos de estratégia: **movimento stutter**, caso este não tenha nenhum inimigo no seu campo de visão e/ou não for "chamado"para ir cercar um inimigo, este dirige-se para a posição mais segura do mapa (o centro do mapa); **movimento ofensivo**, o Líder ordena a sua equipa a cercar um inimigo por ele escolhido, este movimento acontece em qualquer uma das seguintes situações:

- Se no campo de visão do jogador central a sua equipa se encontrar em vantagem numérica, ou seja existirem mais aliados do que inimigos;
- Se no campo de visão do jogador central se encontra em pé de igualdade e este seja do tipo ofensivo.

Por fim, existe o **movimento defensivo**, este movimento consiste no Líder ordenar à sua equipa atacar um inimigo que esteja a atacar o jogador central, sendo que este jogador central tentará fugir. Este movimento ocorre nas seguintes situações:

- Se no campo de visão do jogador central a sua equipa se encontrar em desvantagem numérica, ou seja existirem mais inimigos do que aliados;
- Se no campo de visão do jogador central se encontrar em pé de igualdade e este seja do tipo defensivo. Este pé de igualdade equivale para qualquer número de jogadores (um para um, dois para dois etc).

Em caso de movimento defensivo, o jogador central foge até se encontrar em vantagem, no momento em que no seu campo de visão tenha mais aliados que inimigos ele troca para movimento ofensivo e consequentemente começa a cercar o inimigo também.

Já anteriormente mencionado, estas estratégias, assim como quem ataca são selecionadas pelo líder da respetiva equipa. Assim que o Líder define o inimigo a atacar, este verifica se ele se encontra encostado a uma parede, a um canto ou no meio do mapa, consoante essa informação ele envia o número de jogadores necessários para o cercar e pode ou não enviar mais um aliado para reforçar o ataque (depende da estratégia escolhida).

Por exemplo, no caso em que o inimigo se encontre no canto superior esquerdo (Posição 1,1) o Líder calcula as duas posições adjacentes à posição do inimigo (Posição 2,1 e 1,2) e envia os dois jogadores mais perto dessas posições para o cercar e consequentemente o eliminar, caso os jogadores não cheguem antes do jogador inimigo se mover, o Líder recalcula as posições que os jogadores têm de se posicionar e caso seja necessário envia mais jogadores para apoiar no movimento ofensivo, incluindo um de reforço.

Com o intuito de criar um jogo totalmente imparcial e sem haver a hipótese de uma equipa ganhar por jogar 2 vezes seguidas foi criado a existência de "turnos". Estes turnos são essencialmente a garantia que cada equipa faz apenas uma jogada (o número de jogadores restantes movem-se todos 1 $u.m$), comunica ao árbitro que a sua equipa já jogou, o arbitro recalcula o campo de visão de cada jogador, verifica se alguém da equipa contrária está cercado e envia essa informação ao líder contrário e dá-lhe a vez de jogar. Este processo repete-se até uma das equipas não conter nenhum jogador e o árbitro dar a vitória à equipa que resta.

4 Arquitetura

Um diagrama de classes é uma representação da estrutura e relações das classes. Nesse sentido, passámos por várias fases até obtermos um diagrama de classes final. Para a modelação do nosso sistema multiagente começamos por elaborar um diagrama de classes, presente na Figura 1. Em seguida, explicamos as diferentes classes e as respetivas variáveis.

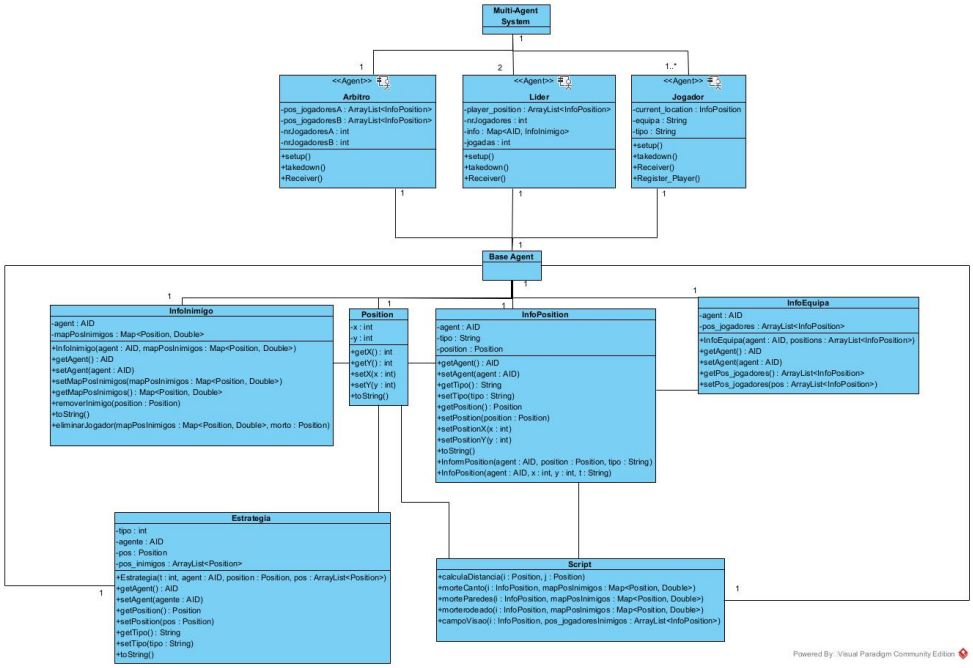

Figura 1: Diagrama de Classe

Como podemos observar, o projeto engloba três agentes, sendo estes o Árbitro, o Líder e o Jogador. O Árbitro, tal como indica o nome, é responsável por verificar as posições dos jogadores de ambas as equipas, e conforme tal calcular o campo de visão respetivo aos jogadores. É ainda responsável por sinalizar a morte de um jogador bem como a troca de turnos. Para tal o Árbitro tem na sua pose constantemente a lista das posições dos jogadores de cada equipa bem como o registo dos turnos/jogadas feitas por cada equipa.

O Líder representa de certa forma um treinador de uma equipa, o Líder não é um jogador mas sim uma entidade à parte que toma decisões quanto às estratégias a adotar conforme o estado do jogo. O Líder é ainda responsável por sinalizar o final do seu turno. Desta forma o Líder tem uma lista das posições dos jogadores da sua equipa, constantemente atualizada, uma mapa com os campos de visão de cada jogador fornecido pelo árbitro, bem como duas variáveis de controlo, a variável jogadas que controla o numero de jogadas feitas para finalizar um turno, e numero de jogadores.

O Jogador, tal como o nome indica, é a entidade responsável por aplicar as estratégias enviadas pelo Líder e dessa forma jogar. Cada Jogador tem a sua própria *InfoPosition* o que no fundo consiste na posição do Jogador, uma equipa que identifica a equipa do Jogador, e por fim um tipo que representa o modo de jogo do Jogador. Este tipo pode ser ofensivo ou defensivo, conforme o seu tipo as suas estratégias de jogo podem variar. Para além dos agentes foram ainda implementadas outras classes que nos permitiram controlar o jogo e garantir a comunicação entre entidades, essas classes são : *Position*, *InfoPosition*, *InfoEquipa*, *InfoInimigo*, *Estrategia*, *Script*.

A classe *Position* representa a posição cartesiana de cada jogador.

InfoPosition faz a ligação de uma Posição a um agente Jogador.

A *InfoEquipa* foi desenvolvida com o propósito de permitir ao Líder comunicar com o Árbitro, esta contém uma lista de todas as *InfoPosition* dos Jogadores da sua equipa.

A classe *InfoInimigo* representa a forma do Árbitro comunicar ao Líder de cada equipa o mapa de inimigos de cada Jogador.

A classe *Estratégia* foi utilizada como meio de comunicação entre Líder e Jogador, onde o Líder envia a cada Jogador o tipo de estratégia que este deve seguir. Se o tipo de estratégia for igual a 0, então o jogador deverá exercer um movimento *stutter*, ou seja, deve movimentar-se para o meio, caso o tipo de estratégia seja igual a 1 então o Jogador deve se movimentar para a posição enviada. Nesta classe é ainda passado um mapa com as posições dos inimigos e respetivos colegas de equipa de forma a que não se passem uns por cima dos outros. Apesar de tal, o mesmo não se verifica uma vez que tivemos algumas dificuldades em implementar essa parte do código.

Por fim temos a classe *Script*, esta classe foi desenvolvida por meio de simplificar o código do Árbitro, assim qualquer função que estivesse a ocupar muito espaço, como era o caso do cálculo de visão e verificação dos diferentes casos de morte de um agente foi realocada para a classe *Script*.

Foi omitido deste diagrama a classe *Mapa*, *MapaDebug*, *RunContainer* e *MainContainer*, uma vez que, não eram relevantes para os *use cases* tratados e assim simplificou o mesmo.

5 Protocolo de Comunicação

Um dos aspetos mais importantes neste projeto é a comunicação entre os diferentes agentes. São estas interações que provocam reações nos agentes e os ajudam a tomar decisões mais eficazes de acordo com um objetivo proposto.

Para representar estas interações foram utilizados diagramas de sequência. Estes diagramas demonstram visualmente as diferentes mensagens que são trocadas entre os agentes, nos diversos momentos da simulação.

Estas comunicações foram baseadas nas normas FIPA utilizadas pelo JADE, ou seja, utilizaremos *performatives* para identificar os vários tipos de pedidos envolvidos entre os agentes.

5.1 Iniciar Arbitro, Líder e Jogadores

No inicio de cada simulação é inicializado um árbitro do jogo. De seguida são inicializados dois líderes, um para a equipa A e outro para a equipa B. Por ultimo são inicializados 5 jogadores com posições, aleatórias ou com um propósito estratégico, possíveis dentro do mapa. Quando os jogadores são criados estes enviam uma mensagem do tipo *SUBSCRIBE* para o Líder da equipa correspondente contendo a sua *InfoPosition*. Após o Líder obter as *InfoPosition* de todos os jogadores este envia uma mensagem do tipo *INFORM* para o Árbitro dando assim entrada da equipa em jogo.

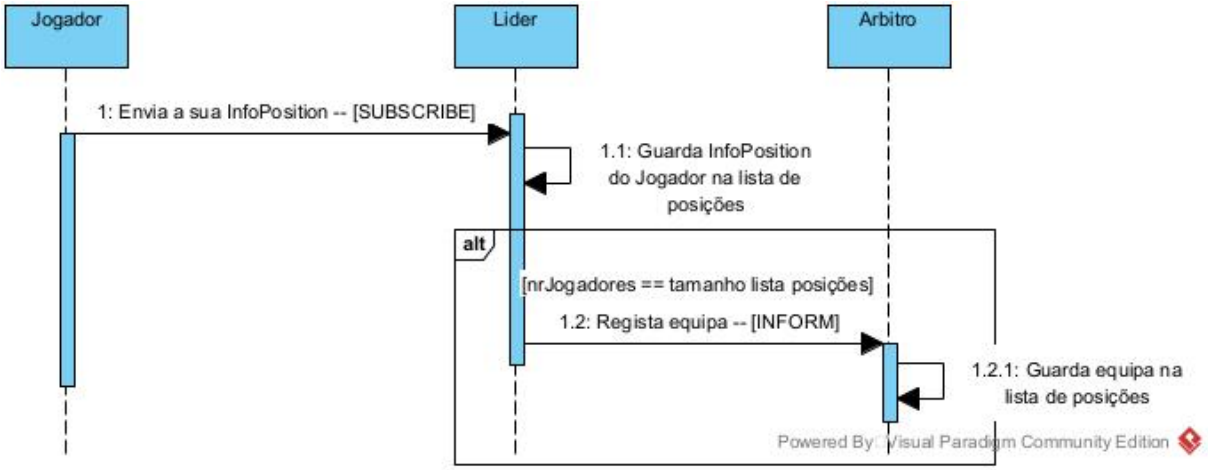

Figura 2: Diagrama de sequencia: inicio do jogo

5.2 Jogador Informa Estado Atual

Durante a simulação o Árbitro e o Líder necessitam de saber a posição dos jogadores. Dessa forma o jogador envia mensagens do tipo *INFORM*, com a sua *InfoPosition*, ao Líder correspondente sempre que realiza um movimento. Quando o Líder recebe o *update* da nova posição do jogador então este envia uma mensagem do tipo *INFORM* para o Árbitro de forma a que este mantenha uma lista constantemente atualizada das posições de todos os jogadores. Sempre que um movimento é feito por parte de um jogador de uma equipa este é sinalizado por parte do jogador com uma mensagem *CONFIRM*. Assim ao fim de recebidas as mensagens *CONFIRM* de todos os jogadores da equipa, o Líder pode sinalizar ao Árbitro o fim do seu turno através de uma mensagem *CONFIRM*.

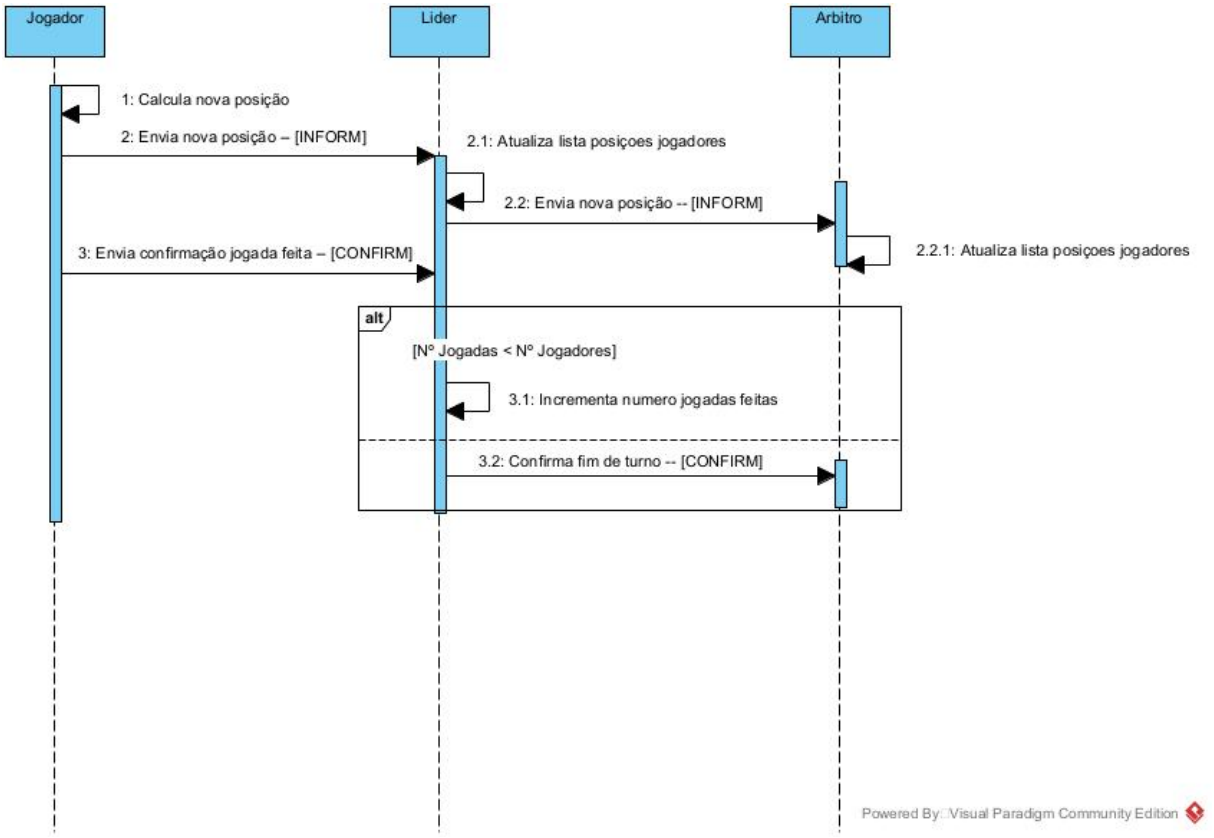

Figura 3: Diagrama de sequencia: *Update* posições do Jogador

5.3 Árbitro Informa Estado Atual

O Líder necessita de saber o estado dos seus jogadores. É o Árbitro que informa ao respetivo líder o número de inimigos no campo de visão de cada um dos seus jogadores. Além disso, é este que verifica se algum jogador da equipa morreu. Desta forma após calcular o campo de visão para todos os elementos de uma equipa, este verifica se algum jogador se encontra encurralado. Se esse for o caso é enviada uma mensagem do tipo *REQUEST* ao Líder respetivo de forma a sinalizar a morte do jogador, para além disso a lista das posições do jogadores no árbitro é atualizada. O Líder por sua vez atualiza a sua lista de posições de jogadores e envia ao respetivo jogador o pedido para morrer através de uma mensagem *REQUEST*. O jogador recetor dessa mensagem procederá com um *doDelete*(), morrendo, e consequentemente saíndo do jogo.

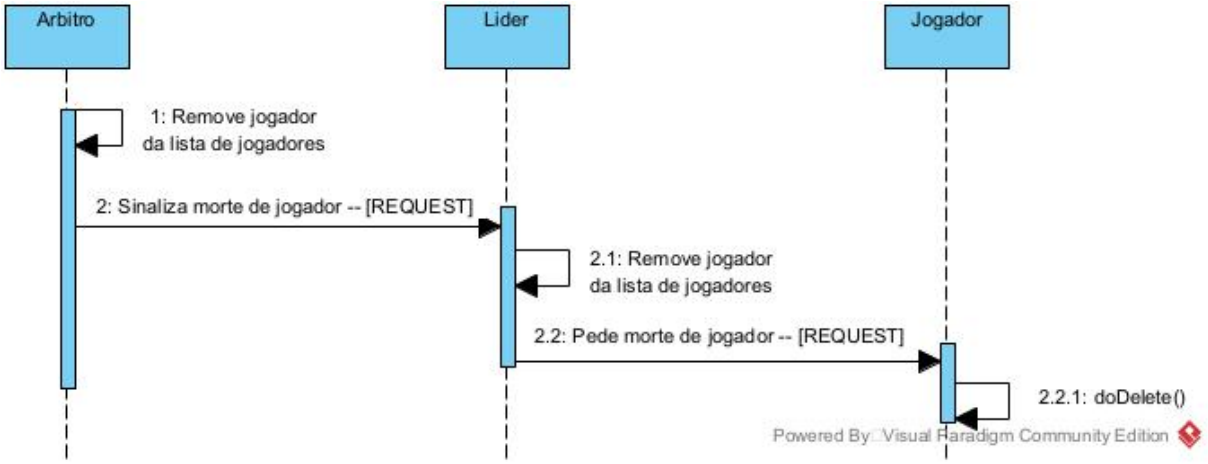

Figura 4: Diagrama de sequencia: Morte de Jogador

No diagrama acima não foi simulado o envio da mensagem do campo de visão uma vez que não se achou necessário para o caso. No entanto essa comunicação será representada em diferentes diagramas ao longo do documento.

5.4 Estratégia Definida pelo Líder

O Líder ao receber a informação relativa ao campo de visão dos seus jogadores calcula a melhor estratégia a seguir. Conforme tal, envia aos seus jogadores a estratégia que devem seguir através de uma mensagem do tipo *INFORM*.

Para calcular a melhor estratégia possível o Líder inicia por determinar o jogador central. Este jogador central será quem tiver mais jogadores à sua volta e o inimigo a atacar será aquele que se encontra mais perto do jogador central. A estratégia escolhida dependerá de quantos aliados/inimigos este jogador central terá no seu campo de visão. Caso este jogador tenha mais inimigos que aliados à sua volta ou esteja em pé de igualdade mas este seja do tipo defensivo ele tem uma estratégia defensiva. Envia os seus aliados contra o inimigos em questão e "foge"até se encontrar em vantagem numérica. Caso o jogador esteja em vantagem numérica ou pé de igualdade mas seja do tipo ofensivo, ele ataca o inimigo juntamente com a equipa.

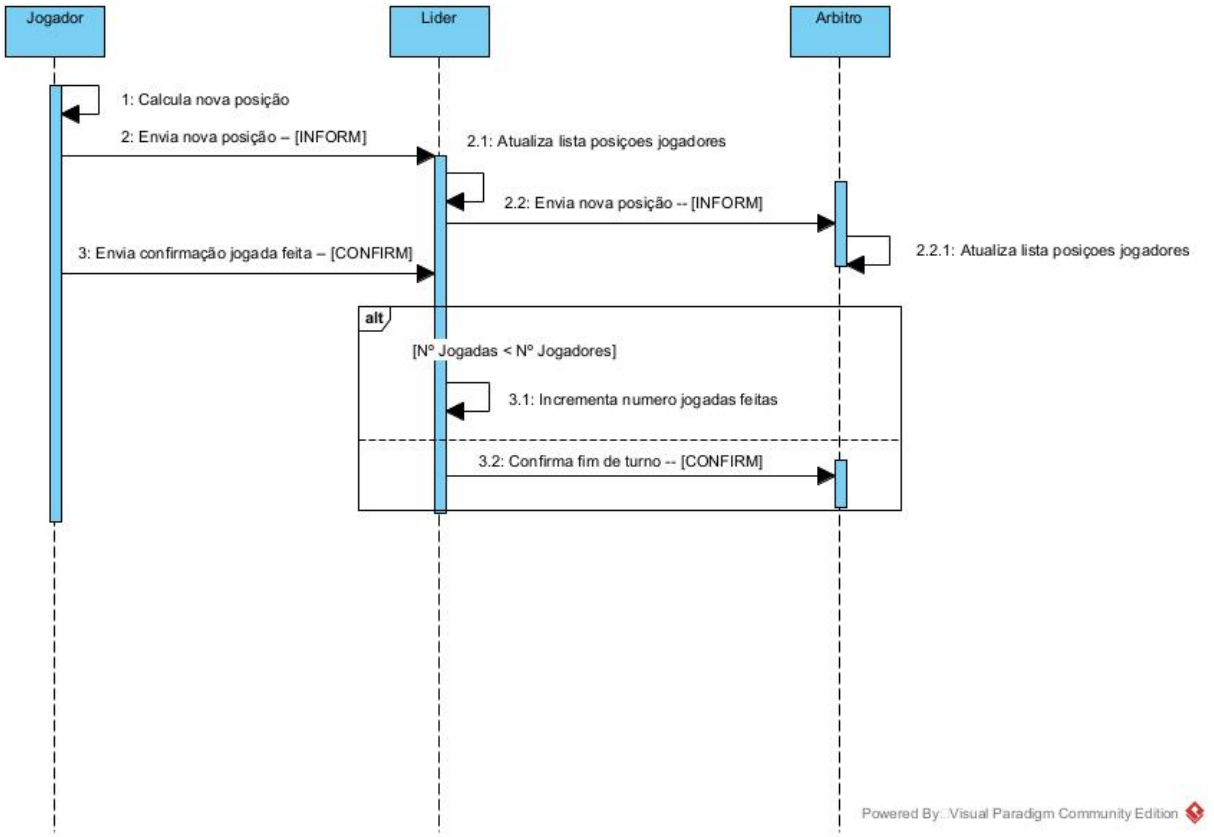

Figura 5: Diagrama de sequencia: Estratégia de jogo

5.5 Diagrama de Comunicação

No diagrama da figura 6 podemos observar como os diferentes tipos de agentes comunicam entre si e internamente, ao longo do jogo. Primeiramente os jogadores dão *subscribe* contendo na mensagem o seu AID e *Position* (*InfoPosition*). Em seguida o Líder regista cada um dos jogadores e no fim regista a equipa no Árbitro. Por sua vez o Árbitro regista as equipas. Em seguida, em cada turno, o Árbitro informa cada um dos Líderes com a informação do campo de visão dos seus jogadores e avisa caso algum jogador morra. Por sua vez cada Líder anota as posições dos inimigos, cria uma estratégia para cada jogador e envia-a. Por sua vez o jogador vai-se mover conforme a estratégia recebida e informar o Líder, que acaba o turno entregando as posições atualizadas dos seus jogadores ao Árbitro.

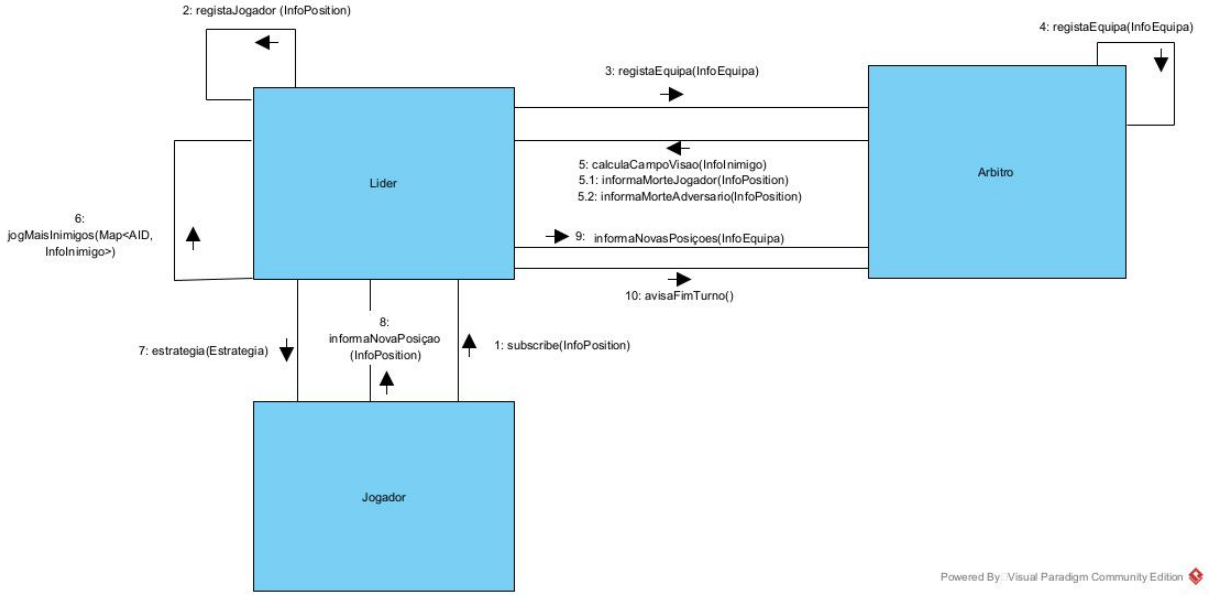

Figura 6: Diagrama de comunicação

6 Processamento Interno dos Agentes

Nesta secção apresenta-se os diagramas relativos ao processamento interno de cada um dos agentes envolvidos no sistema.

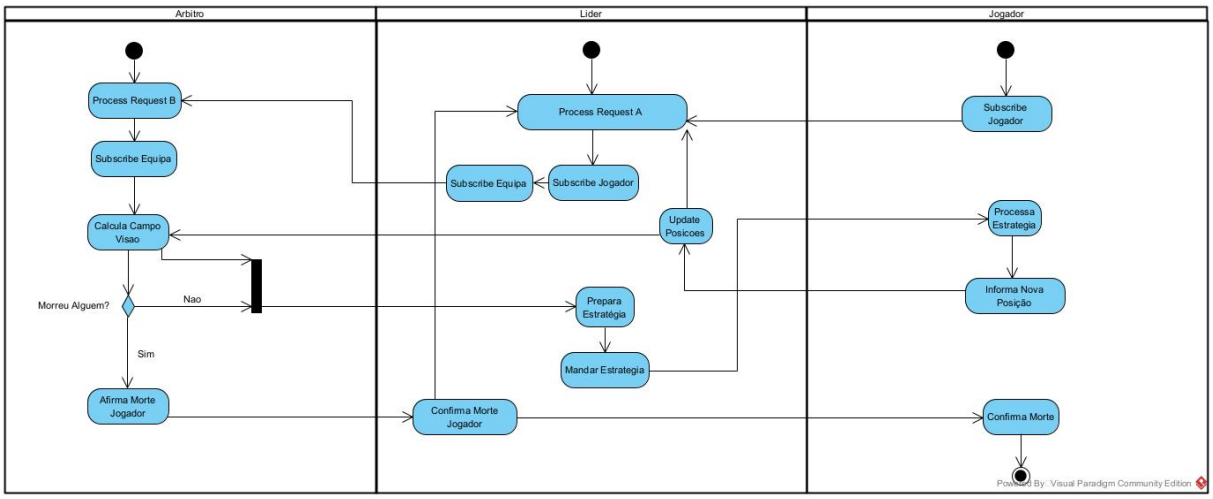

Figura 7: Diagrama de Atividades

6.1 Agente Jogador

Tal como pode ser visto na figura 7, o agente jogador faz *subscribe* ao Líder da sua equipa, e fica à espera de uma posição para qual se deslocar (movimento *stutter* ou

posição adjacente a um inimigo). A cada movimento (1 $u.m$) ele informa o Líder da sua nova posição. Este está ativo até ordem contrária do Árbitro, que indica se o mesmo foi cercado ou não pelos inimigos. Quando o jogador morre, o agente termina.

6.2 Agente Líder

O Líder, após iniciar o programa regista os jogadores da sua equipa apenas uma vez e a cada turno atualiza as posições dos seus jogadores e envia essa informação ao agente Árbitro. A cada turno, O Líder recebe a informação proveniente do árbitro, nomeadamente se algum da sua equipa está cercado e em que posição os inimigos se encontram no campo de visão dos seus jogadores. A partir desta informação ele parte para o estado de determinar qual o inimigo a atacar, se o jogador central deve ou não atacar, as posições necessárias a ocupar e determina o jogador mais perto de cada posição e envia o mesmo para lá. Este comportamento ocorre em cada ciclo até não haver mais jogadores na sua equipa.

6.3 Agente Árbitro

Este agente é iniciado sempre que se inicia o programa. Este agente inicia por registar as equipas e depois repete sempre pela mesma ordem os seguintes estados. Começa por calcular o campo de visão de cada aliado e se tiver, anota as posições dos inimigos em relação a um especifico aliado. De seguida passa para o estado de verificar se alguém aliado está cercado e caso seja verdade, anota para indicar ao Líder da respetiva equipa. Assim que termina estes dois estados dá a vez de jogar ao Líder em questão. Estes estados repetem-se até que uma das equipas não contenha mais nenhum jogador.

7 Implementação dos Agentes

Arquiteturalmente foram definidas uma série de diretrizes que os agentes têm de seguir:

- *Behaviours*:

  - Todos os agentes têm um comportamento responsável pela receção e tratamento de mensagens, trata-se de um comportamento cíclico;

  - O agente Jogador tem um comportamento associado para informar a sua criação/existência no jogo, sendo que o envio dessa mensagem inicial ao agente Líder contendo a sua posição foi tratada com um comportamento único, executado no início da simulação.

- As mensagens enviadas entre os todos os agentes do sistema contém Objetos para transportar a informação, sendo que estas também são únicas e distintas para as diferentes ações no sistema.

7.1 Árbitro

O Árbitro é o agente que tem conhecimento acerca de tudo o que se passa no jogo. É ele que contém a informação respetiva as posições dos jogadores de ambas as equipas e com isso controla as mortes dos agentes e os turnos. Para além disso este é capaz de desencadear um conjunto de ataques, conforme o cálculo dos campos de visão dos jogadores. Esta classe contém um único método fulcral sendo este um comportamento cíclico, neste, o arbitro recebe a informação das equipas e toma decisões quanto a quem deve jogar. É neste método também que o campo de visão dos jogadores é calculado e são sinalizadas a morte de jogadores.

Inicialmente o árbitro começa por receber uma mensagem de *INFORM* dos líderes das equipas A e B, com a informação relativa as posições dos jogadores de cada equipa, conforme tal este guarda as mesmas num *array* respetivo. Registadas as equipas o Árbitro declara a vez da equipa A jogar. Esta equipa é sempre a primeira a começar o jogo. Desta forma o árbitro com auxílio da classe *Script* calcula o campo de visão para todos os jogadores da equipa A. De seguida verifica se, dadas as posições iniciais, nenhum jogador se encontra desde logo rodeado. Caso tal não se verifique este envia uma mensagem do tipo*INFORM* para o Líder contendo o mapa de inimigos no campo de visão de cada jogador respetivo da equipa A. Caso algum jogador tenha de facto morrido, então uma mensagem do tipo *REQUEST* será enviada para o Líder da equipa correspondente, de forma a retirar o jogador morto do jogo. Nesse momento será atualizada a lista das posições no árbitro. É simultaneamente enviada uma mensagem para o Líder da equipa adversária do tipo *AGREE* que anuncia a morte do jogador adversário para que este possa atualizar as suas listas de inimigos.

Terminada a vez da equipa A, o árbitro será sinalizado pelo Líder correspondente através de uma mensagem do tipo *CONFIRM*, assim este dará a vez a equipa B de jogar, repetindo todo o processo acima descrito.

Por fim, ao longo do jogo vão sendo contadas o numero de jogadas feitas por equipa de forma a conseguir medir o grau de eficácia das estratégias.

7.2 Líder

O Líder é o agente que tem conhecimento acerca de tudo o que se passa relativamente à sua equipa. É ele que contém a informação relativa às posições dos jogadores bem como os mapas dos seus inimigos. É este que atribui as tarefas a cada um dos agentes Jogador para o desenvolvimento do jogo. Esta classe contém um único método fulcral sendo este um comportamento cíclico. Neste o Líder recebe a informação dos jogadores bem como informações do árbitro, as quais utiliza de forma a tomar decisões quanto à estratégia a seguir. É neste método também que o Líder sinaliza o final do seu turno e avisa jogadores da sua morte.

Inicialmente o Líder começa por receber varias mensagens do tipo *SUBSCRIBE* dos jogadores da sua equipa contendo informação quanto a sua existência e as suas posições. Após receber todas as mensagens *SUBSCRIBE* este envia uma mensagem do tipo*INFORM*

para o árbitro de forma a registar a sua equipa.

Após o registo da sua equipa este aguarda a sua vez de jogar. Sendo o seu turno, o Líder recebe uma mensagem do tipo*INFORM* proveniente do árbitro contendo informação quanto aos inimigos no campo de visão dos seus jogadores. Conforme tal este irá preparar a melhor estratégia a adotar tendo em conta o estado atual do jogo. A forma como a estratégia é escolhida pode ser melhor compreendida na secção acima, no agente Líder. Uma vez escolhida a estratégia será enviada uma mensagem do tipo *INFORM* aos jogadores contendo a mesma. Caso o jogador receba uma estratégia de tipo 0, este deverá efetuar o movimento *stutter*, ou seja, deve-se movimentar para o centro do campo, caso contrário (estratégia do tipo 1) este deve movimentar-se em direção a posição escolhida pelo árbitro. Após cada movimento feito pelos jogadores será recebida uma mensagem *CONFIRM* por parte dos mesmos controlando assim o número de jogadas feitas. Quando o numero de mensagens recebidas equivale ao número de jogadores então o Líder envia uma mensagem do tipo *CONFIRM* para o árbitro sinalizando então o fim do seu turno.

Quando o Líder recebe a mensagem com os mapas de inimigos dos jogadores pode receber simultaneamente mensagens do tipo *REQUEST* e *AGREE*. Caso este receba uma mensagem do tipo *REQUEST* este irá remover da sua lista de jogadores o jogador que morreu e em seguida vai enviar uma mensagem do tipo *REQUEST* ao jogador respetivo para que este efetue a sua terminação. Caso receba uma mensagem do tipo *AGREE*, este foi notificado da morte de um jogador adversário, nesse caso ele deverá retirar da lista de inimigos esse mesmo jogador.

7.3 Jogador

Sendo que o objetivo de um jogador é matar um jogador inimigo, este tem de se saber movimentar pelo mapa. Desta forma o jogador inicialmente envia uma mensagem do tipo *SUBSCRIBE* ao Líder da sua equipa contendo informação sobre a sua posição inicial, uma vez que é o líder que tem conhecimento sobre qual o próximo movimento do mesmo. Após enviada esta mensagem o jogador aguarda por uma mensagem do tipo *INFORM* do Líder, essa mensagem contem a estratégia que o jogador deve usar no próximo passo. Assim sempre que este recebe uma mensagem desse tipo o mesmo preocupa-se somente em se movimentar. Consequentemente este envia uma mensagem do tipo *INFORM* para o Líder juntamente com uma mensagem do tipo *CONFIRM*. Na primeira mensagem este envia a sua nova posição no mapa garantindo a atualização do mapa das posições do Líder, enquanto que na segunda este confirma a sua jogada permitindo ao líder controlar o numero de jogadas já feitas antes de terminar o turno. O jogador poderá ainda receber uma mensagem do Líder do tipo *REQUEST*. Nesse caso o jogador é sinalizado da sua morte e deverá efetuar a sua terminação.

8 Avaliação e testes de performance dos agentes

8.1 Ferramenta de visualização

De modo a ser possível uma visualização mais apelativa dos agentes criados, foi realizada uma interface gráfica que acompanha os jogadores ao longo dos turnos. Essa visualização passa por um mapa de dimensão 35 por 35 quadrados correspondentes as posições possíveis dos jogadores. De modo a deixar a visualização apelativa e não apenas informativa foram usadas texturas tanto para as posições como para os jogadores. Assim para o terreno, foram usadas duas texturas diferentes de areia, para dar um aspeto tipo xadrez, e para as equipas foram desenhados soldados, uns azuis e outros vermelhos, para simbolizarem as duas equipas.

Figura 8: Textura 1 de areia

Figura 9: Textura 2 de areia

Figura 10: Jogador da equipa azul

Figura 11: Jogador da equipa vermelha

O mapa resultante da junção das texturas anteriores com os seus respetivos objetos pode-se ver em seguida :

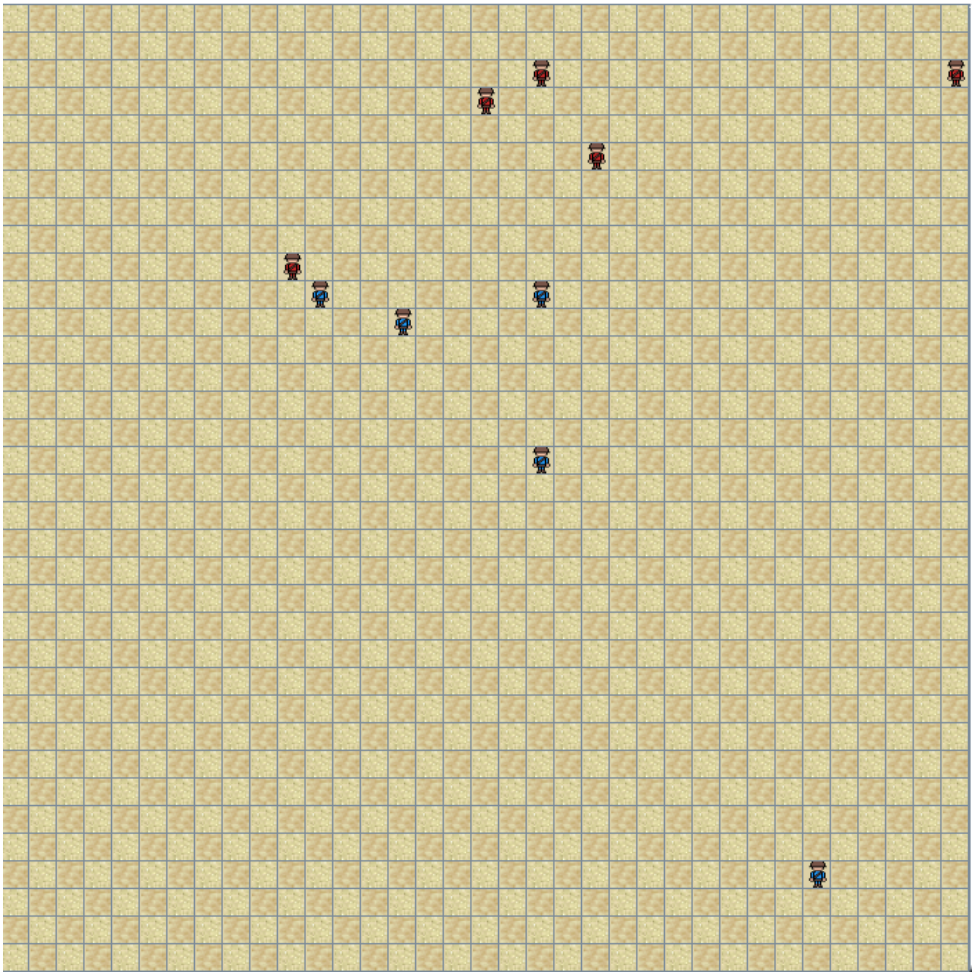

Figura 12: Mapa de jogo

8.2 Funções Objetivo e Análise de Estratégias

De modo a poder avaliar a performance dos agentes desenvolvidos foram criadas duas funções objetivo, sendo que o x representa o número de jogadores matados pela equipa, o y o número de jogadores da equipa que morreram e z o número de turnos até acabar o jogo. A primeira função pensada para mostrar que matar inimigos era positivo e que tanto morrerem aliados como um número excessivo de turnos era negativo, encontra-se em seguida.

$$score = 1000x - 500y - 5z$$

A função objetivo não serve simplesmente para podermos olhar para ela e dizermos que uma determinada estratégia obteve um melhor *score* que outra. Poderia ser usada num trabalho futuro para ajudar um modelo de *reinforcment learning* ou de *q learning* a aprender a jogar com base nesse *score final*. Assim faria mais sentido que a função objetivo não desse valores inferiores a 0, sendo que surgiu a próxima função:

$$score = (1000x + 500y)/z$$

Com esta função objetivo resultante já podemos medir um pouco melhor a qualidade das estratégias desenvolvidas, sendo que, caso fosse usada para aprendizagem automática, provavelmente seria necessário realizar um *tunning* aos parâmetros utilizados. No entanto já nos serviu para poder avaliar as nossas duas estratégias de jogadores, a ofensiva e a defensiva. Através dos testes realizados foi-nos permitido concluir que uma equipa de jogadores ofensivos é melhor do que a opção defensiva, isto porque a equipa que elimina primeiro um inimigo é normalmente aquela que fica em vantagem para ganhar o jogo. No caso em que a equipa defensiva ganhava, também era prejudicada face ao fator número de turnos. Esta estratégia de fugir, levava a que fosse necessário um maior número de turnos para poder eliminar os inimigos, de modo que a nossa função acabava por beneficiar a estratégia ofensiva.

Na tabela da figura 13 podemos observar o resultado de 15 jogos com posições iniciais aleatórias entre as equipas A e B, sendo que a equipa A era totalmente defensiva e a B era totalmente ofensiva. Como podemos observar o que foi dito anteriormente confirma-se. A equipa ofensiva ganha em 10 dos 15 jogos sendo o seu *score* médio de 152 e o número de turnos médio igual a 51 turnos. Já a equipa defensiva ganhou apenas 5 dos jogos, obtendo um *score* médio de 138 e um número médio de turnos de 60 turnos.

Decidimos ainda fazer um teste com uma equipa mista, ou seja, a equipa B, toda atacante e uma nova equipa A, que desta vez tinha uma mistura de jogadores ofensivos e defensivos. Os resultados deste teste encontram-se na figura 14. Podemos ver que, embora já não com tanta vantagem, a equipa totalmente atacante continua a ser a melhor opção. Assim em 10 tentativas realizadas em mapas aleatórios a Equipa B (totalmente ofensiva) obteve 60% de vitórias, com média de 115 de *score* e média de 70 turnos. Já para a equipa A obteve 40% de vitórias com média de *score* igual a 114 e média de turnos de 66.

Equipa Vencedora	Número de turnos	Score
Equipa B	42	176
Equipa B	34	220
Equipa B	57	131
Equipa A	52	144
Equipa A	70	107
Equipa B	45	166
Equipa B	51	147
Equipa A	63	119
Equipa B	52	144
Equipa B	71	105
Equipa A	85	88
Equipa B	67	111
Equipa B	43	174
Equipa B	52	144
Equipa A	32	234

Figura 13: Resultados jogos entre equipas totalmente ofensivas e totalmente defensivas

Equipa Vencedora	Número de turnos	Score
Equipa B	82	91
Equipa A	64	117
Equipa B	80	93
Equipa B	96	72
Equipa B	77	90
Equipa A	61	122
Equipa A	78	96
Equipa A	61	122
Equipa B	43	174
Equipa B	44	170

Figura 14: Resultados jogos entre equipas totalmente ofensivas e equipa mista

9 Conclusão e Trabalho Futuro

Dado por concluído o trabalho entendemos que conseguimos alcançar o objetivo principal do projeto que era conceber e implementar um sistema multiagente, com agentes inteligentes capazes de cooperar para atingir um objetivo.

Apesar do projeto concluir os objetivos principais e regras definidas, apresenta uma falha que deve ser resolvida em trabalho futuro. Durante os movimentos os jogadores acabam por se sobrepor de modo a irem para o caminho mais perto que os líder lhe ordena. Na nossa implementação o líder aponta a posição para a qual o jogador deve ir, e este é que define o seu caminho. De modo a resolver este problema chegámos a implementar um algoritmo que se encontra nos anexos, que basicamente pegava na informação das posições ocupadas fornecida pelo líder, e criava uma matriz de *char* assinalando com 'S' a sua posição, a '1' as posições livres, a '0' as posições ocupadas e a 'D' os destino. Depois recorrendo a um algoritmo de pesquisa *breadth-first* encontrava o caminho mais perto de 'S' a 'D' e o jogador dava o passo para a posição seguinte fornecida pelo mesmo. Esse código embora funcionasse para um número de rondas, chegava sempre uma altura em que ocorriam *leaks* de memória. Nos anexos encontra-se um link para um vídeo de demonstração do algoritmo em prática enquanto funcionava. Uma vez que não estava explícito no relatório a necessidade dos jogadores não se sobreporem, acabámos por deixar este problema um pouco de lado, e no fim acabámos por não o conseguir resolver.

Em relação às implementações mencionadas/descritas no relatório, podemos afirmar que todas foram realizadas com sucesso. Sendo que a parte mais complicada foi manter a persistência e coerência dos dados no momentos das comunicações entre os diferentes agentes. Outro problema bastante corrente foi a correção de comportamentos indefinidos (exceções) por entre os vários.

Como trabalho futuro o primeiro foco seria resolver este problema, e em seguida tentar utilizar a função objetivo criada de modo a treinar modelos de aprendizagem automática a tomarem decisões ainda mais inteligentes do que aquelas pensadas pelo grupo.

Embora com o percalço acima referido, consideramos que o balanço do trabalho é positivo. Este trabalho permitiu-nos não só perceber os mecanismos intrínsecos de sistemas multi-agente, mas também ficar familiarizados com a biblioteca JADE, de modo a já estarmos preparados para o futuro, caso que seja necessário trabalhar com as mesmas ferramentas de novo.

10 Anexos

10.1 Código de cálculo do caminho mais curto

```
class Cello {
    int row;
    int col;
    public Cello(int rowIndex, int colIndex) {
        super();
        this.row = rowIndex;
        this.col = colIndex;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Cello cell = (Cello) o;
        return row == cell.row &&
                col == cell.col;
    }

    @Override
    public int hashCode() {
        return Objects.hash(row, col);
    }
}



public class ShortestPath {

    private static List<int[]> shortestPath(char[][] grid) {
        ArrayList<int[]> path = new ArrayList<>();
        for (int i = 0; i < grid.length; ++i) {
            for (int j = 0; j < grid[0].length; ++j) {
                if (grid[i][j] == 'S') {
                    bfs(grid, new Cello(i, j), path);
                }
            }
        }
        return path;
    }

    public static Position move(Estrategia e, Position jogador) {

        char[][] mapa = new char[35][35] ;
```

```
        for (int i = 0 ; i < 35 ; i++) {
            for (int j = 0 ; j < 35 ; j++) {
                mapa[i][j] = '1';
            }
        }

        ArrayList<Position> bloqueado = e.getPos_Inimigos();

        for (Position b : bloqueado) {
            mapa[b.getX()-1][b.getY()-1] = '0';
        }

        mapa[jogador.getX()-1][jogador.getY()-1] = 'S';

        mapa[e.getPosition().getX()-1][e.getPosition().getY()-1] = 'D';

        List<int[]> caminho = shortestPath(mapa);

        System.out.println("ENTRA AQUI E NAO FAZ");

        caminho.stream().forEach(i -> {
            System.out.println("{" + i[0] + "," + i[1] + "}");
        });

        if(caminho.size()>1) {
            int[] novaPosicao = caminho.get(1);

            Position res = new Position(novaPosicao[0] + 1, novaPosicao[1] + 1);
            return res;
        } else {
            return jogador;
        }
    }

    private static void bfs(char[][] grid, Cello start, List<int[]> path) {

        int[] xDirs = new int[] {0,0,1, -1};
        int[] yDirs = new int[] {1,-1, 0, 0};

        Queue<Cello> bfsQueue = new LinkedList<>();
        bfsQueue.add(start);
        HashMap<Cello, Cello> parentMap = new HashMap<>();
        boolean[][] visited = new boolean[grid.length][grid[0].length];
        Cello endCell = null;
        while(!bfsQueue.isEmpty()) {
            boolean flag = false;
            Cello from = bfsQueue.poll();
```

```
            for (int k = 0; k < xDirs.length; ++k) {
                int nextX = from.row + xDirs[k];
                int nextY = from.col + yDirs[k];

                if (nextX < 0 || nextX >= grid.length || nextY < 0
                        || nextY >= grid[0].length || grid[nextX][nextY] == '0'
                        || visited[nextX][nextY]) {
                    continue;
                }

                visited[nextX][nextY] = true;
                Cello nextCell = new Cello(nextX, nextY);
                bfsQueue.add(nextCell);

                parentMap.put(nextCell, from);

                if (grid[nextX][nextY] == 'D') {
                    endCell = new Cello(nextX, nextY);
                    flag = true;
                    break;
                }
            }
            if (flag) {
                break;
            }
        }
        Stack<Cello> stack = new Stack<>();
        stack.push(endCell);


        while (true) {
            Cello fromCell = parentMap.get(endCell);
            stack.push(fromCell);
            if (fromCell == start) break;
            endCell = fromCell;
        }
        while (!stack.isEmpty()) {
            Cello p = stack.pop();
            path.add(new int[] {p.row, p.col});
        }
    }
}
```

10.2 Demonstração do código acima

https://youtube.com/shorts/xnR9z-hsWIw?feature=share