Mestrado em Engenharia Biomédica – Informática Médica

1º Ano | 2022/2023 | 1º semestre

Sistemas Inteligentes

Trabalho Prático de Grupo

Grupo 4

Docentes:

Paulo Novais

Filipe Gonçalves

Ciarán McEvoy A87240

Hugo Silva PG50416

Gonçalo Carvalho PG50392

Tomás Lima PG50788

Índice

Introdução

A criação de Sistemas Inteligentes (SI) é hoje, e cada vez mais, fulcral no processo evolutivo da sociedade. Numa cidade cosmopolita atual, a sua presença passa já despercebida, desde os semáforos reguladores de trânsito aos robôs de cozinha, tendo já substituído certas profissões que se consideravam tradicionais. Mas o seu papel não é já apenas preponderante nos centros urbanos. Em grandes explorações agrícolas, o papel de alfaias agrícolas controladas por Inteligência Artificial (IA) e sistemas distribuídos é essencial para se atingirem as cotas de produção necessárias para alimentar a população mundial.

O presente trabalho visa o planeamento e implementação de um SI que simule um "jogo de estratégia", no qual os agentes de uma equipa têm como objetivo cooperar entre si para eliminar os agentes da equipa adversária. Será então essencial conferir a estes agentes a capacidade de interpretação daquilo que os rodeia no seu mundo, de comunicação com os seus pares e de definição de métodos de cooperação estruturados de forma a atingir o objetivo.

Definição do Problema

O jogo ocorre num espaço bidimensional de dimensão 100x100, no qual existem 2 equipas com 5 agentes cada, que têm o objetivo de explorar o mundo ao seu redor e eliminar os agentes da equipa adversária, com uma visão limitada de 7x7, podendo, para tal, mover-se em 1 unidade para as posições adjacentes de cada vez.

Os agentes são eliminados se estiverem cercados em todas as 4 posições adjacentes à sua (em cima, em baixo, à esquerda e à direita), tanto pelos inimigos, como pelos próprios amigos ou pelos limites do espaço.

Assim, em cada turno, cada agente da equipa que está a jogar tem de escolher 1 de 5 movimentos (ações) possíveis: ficar no sítio em que está ("stay"), ir para cima ("up"), ir para baixo ("down"), ir para a esquerda ("left") ou ir para a direita ("right"). Depois de todos se moverem, avança o turno para a vez da outra equipa.

Desta forma, os agentes terão de comunicar entre si para conseguir cooperar e ganhar o jogo, que se consegue quando se elimina 4 dos 5 adversários.

Bibliotecas SPADE e PyGame

SPADE (*Smart Python Agent Development Environment*) é uma plataforma multiagente escrita em python, utilizada na implementação de agentes inteligentes, que faz uso de um sistema de mensagens instantâneas (XMPP). A arquitetura SPADE é composta por agentes que permanecem num servidor XMPP, em que cada agente contém um ID próprio. Esta plataforma faz uso de especificações FIPA de modo a permitir o desenvolvimento de agentes que conseguem comunicar entre si e com outros humanos tornando possível a interoperabilidade de um sistema multiagentes.

Para a criação do *display* do jogo, foi utilizado o package PyGame. Através deste package foi possível criar uma janela separada, de tamanho pré-definido, em que os agentes são todos colocados no tabuleiro com a cor da sua respetiva equipa, como se verifica na Figura 1.

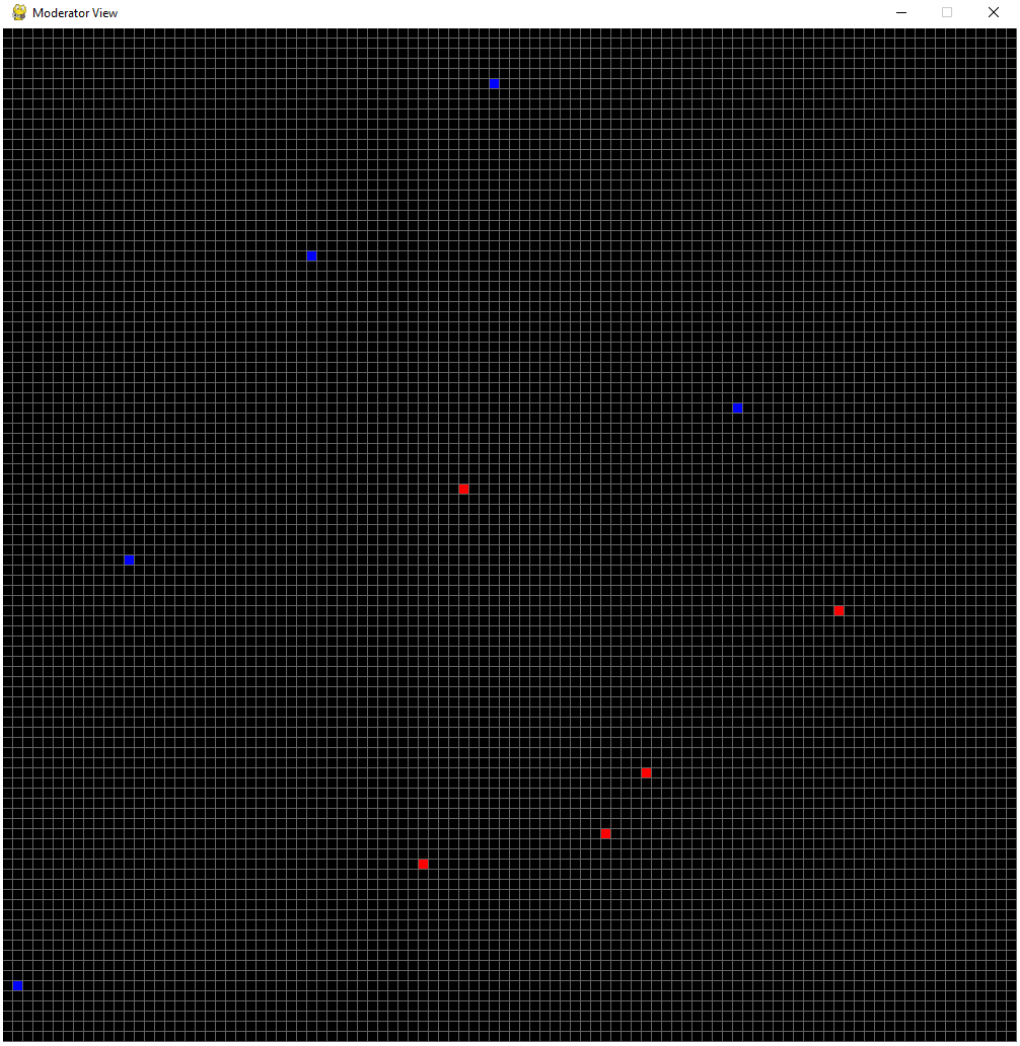

Figura 1 – *Display* do jogo, criado através da biblioteca PyGame.

Planeamento e Estruturação

O primeiro passo deve ser sempre o de definir de que forma se vai estruturar o sistema e a comunicação entre agentes, antes de se iniciar a escrita do código. Isto permite evitar a reescrita de secções à *posteriori* devido a erros de planeamento e torna mais expedita a construção deste.

Existem, para a cooperação, dois tipos de planeamento, o central e o distribuído. Após alguma ponderação de ambos os planeamentos, optou-se pelo planeamento central, ou seja, apenas um agente seria responsável por executar a estratégia de toda a equipa. Deste modo, foram criados três tipos de agentes, sendo eles: Moderador (responsável pelo cumprimento das regras do jogo), Comandante (o "cérebro" da equipa, responsável por definir os movimentos dos seus elementos) e Soldado (as "peças" que formam a equipa).

Os diagramas de classes de atividades destes agentes encontram-se no **Anexo 1**.

Além disto, foram também decididos diversos *checkpoints* ao longo do percurso para testar o funcionamento devido do programa:

- Depois da criação dos agentes e de um tabuleiro (*display*), a primeira fase de testes foi a de ver se era possível observar os soldados a serem colocados no tabuleiro, sem nenhum tipo de movimento, testando assim o registo de agentes e a inicialização do jogo.
- A fase seguinte foi a da criação de um algoritmo de movimentos completamente aleatórios, sendo assim testada a capacidade de receber e enviar mensagens entre os agentes, assim como o funcionamento de vertentes específicas, tais como a eliminação de soldados, a consistência e o respeito pelas regras do jogo, entre outras.
- Provado isto, o passo seguinte foi o da criação de um algoritmo mais elaborado, começando por desenvolver a estratégia de ataque, mantendo o método de procura aleatória. O principal objetivo desta fase foi a aquisição de informação acerca do posicionamento dos inimigos e melhorar o algoritmo da sua captura, assim como detetar erros que tivessem passado as fases de teste anteriores.
- Para terminar a fase de desenvolvimento, bastou incorporar ao algoritmo anterior uma procura planeada e, por fim, testar a cooperação e o funcionamento da estratégia final tendo em conta os parâmetros definidos. Foram efetuados diversos jogos nos quais os agentes, munidos das estratégias para maximizar o seu desempenho, se enfrentaram uns aos outros.

Classes

Para uma melhor e mais fácil transmissão e tratamento dos dados, foram criadas as seguintes classes:

- ***Board***

Contém 2 atributos, a lista de listas *matrix* e o dicionário *teams*. Esta classe implementa o sistema de coordenadas bidimensional no atributo *matrix*, em que para obter o valor de um ponto nas coordenadas (x, y) é necessário fazer *matrix[y][x]*, visto que cada linha na *matrix* corresponde a um y, e cada coluna a um x. Para além disso, também é utilizado pela biblioteca PyGame para o *display*, mostrando a informação da *matrix* de acordo com as cores das equipas, conseguidas através do dicionário *teams*.

- ***Coordinates***

Contém 2 números inteiros, que correspondem às coordenadas (x, y) de um ponto no espaço bidimensional.

- ***Infosoldado***

Contém informação relativa ao agente Soldado, ou seja, o seu JID, a sua equipa, e as suas coordenadas (através da classe *Coordinates*).

- ***Infocommander***

Contém informação relativa ao agente Comandante: o seu JID e a sua equipa.

Atributos e *Behaviours* dos Agentes

Soldado

Cada soldado guarda o seu JID, equipa, e as suas coordenadas através da classe *InfoSoldado*. Este guarda, na lista *vision*, a informação sobre o ambiente que o rodeia (*InfoSoldados* presentes no campo de visão 7x7), para envio ao comandante, que tomará as decisões baseando-se nesta, enviada por cada um dos seus soldados vivos.

Estes agentes implementam apenas 2 behaviours: *SoldierRegister()* e *SoldierReceive()*, que iniciam ambos a par do soldado.

- ***SoldierRegister()***

Este behaviour, do tipo *OneShotBehaviour*, faz com que o soldado envie uma mensagem, com a performative *register*, ao moderador contendo os seus dados (*InfoSoldado*), como forma de indicar que quer jogar.

- ***SoldierReceive()***

É um *CyclicBehaviour* que permite que o soldado consiga receber e tratar as ordens do moderador e do comandante. Pode receber mensagens com quatro *performatives* distintas, sendo elas: *inform*, *your_turn*, e *order_kill*, vindas do moderador, e *movement_order*, enviada pelo seu comandante.

Caso seja recebida uma mensagem com a *performative inform*, esta contém uma lista dos agentes que estejam no seu campo de visão. Ao percorrer a lista, se o soldado identificá um *InfoSoldado* com o seu JID, mudando a sua posição para as coordenadas desse *InfoSoldado*. No caso de serem identificados outros agentes que não ele próprio no campo de visão, estes serão adicionados à sua visão pessoal.

No caso da *performative* recebida ser *your_turn*, esta é uma indicação de que é a vez da sua equipa jogar. O soldado envia então uma mensagem ao seu comandante, com a *performative soldier_intel*, em que é enviada uma lista com os seus dados, utilizando a classe *InfoSoldado*, e a sua visão do campo.

Se a mensagem recebida tiver a *performative movement_order*, o soldado recebe do comandante uma ordem de movimento. Consequentemente, o soldado redireciona a mensagem ao moderador, com a *performative ask_move*, enviando uma lista contendo os seus dados através da classe *InfoSoldado*, para identificação, e a direção que lhe foi dada para se movimentar.

Por fim, se a *performative* da mensagem for *order_kill*, esta é uma informação do moderador a avisar que ele morreu, e, portanto, o soldado envia uma mensagem com a *performative died* ao seu comandante para o notificar desse acontecimento e faz *stop()* ao agente.

Moderador

O moderador pode ser visto como uma espécie de comando central, que tem como principal função assegurar um funcionamento correto do jogo, assim como das regras que se pretende que sejam seguidas. É este agente que trata da eliminação de soldados que tenham sido "mortos", assim como de verificar se o jogo já terminou ou não, da inicialização quer dos turnos, quer do jogo em si mesmo, entre outras tarefas de manutenção.

Assim sendo, o nosso moderador necessita de ter informação acerca de tudo o que se passa, e por isso, terá acesso a recursos específicos, sendo estes: as dimensões do tabuleiro (x_*max* e y_*max*, inteiros), o tabuleiro do jogo (board, da classe *Board*), o número da jogada atual (*turn*, inteiro) e atributos responsáveis pela gestão das equipas.

O nome das equipas são *strings*, mas é também interessante que possam ser representadas por inteiros, de forma a facilitar alguns cálculos. Desta forma, o dicionário *registered_teams_dic* permite passar o nome da equipa de *string* (*key*) para o seu inteiro (*value*), a lista *registered_teams_list* permite fazer o oposto, passar o seu inteiro (*index*) para a *string* (*value*), a lista de listas (matriz) *registered_soldiers_info* permite obter a lista de soldados vivos (*value*) de uma determinada equipa (*index*). A variável inteira *playing_team* permite, em conjugação com a matriz *registered_soldiers_info*, obter e monitorizar todos os soldados que estão a jogar.

Este agente tem 4 *behaviours*, que têm uma ordem específica de invocação, 2 deles responsáveis pelo início do jogo (*ModeratorRegister()* e *SetupGame()*) e o os outros 2, que funcionam à vez, de forma "cíclica", que efetuam cada turno do jogo (*IniciateNextTurn()* e *PrepareNextTurn()*).

- ***ModeratorRegister()***

Este é um *CyclicBehaviour*, responsável por registar todas as equipas e soldados em jogo nos atributos respetivos do agente (*registered_teams_dic*, *registered_teams_list* e *registered_soldiers_info*), recebendo para tal mensagens com a *performative register*.

Quando recebe uma mensagem de um soldado de uma equipa não registada (não existe nas *keys* do *registered_teams_dic*), atualiza todas as variáveis referidas, atribuindo

um inteiro a essa equipa (começando por 0, e aumentando em 1 por cada equipa), e adiciona-o à sua respetiva lista na matriz *registered_soldiers_info*.

Passado um certo tempo sem receber mensagens (tempo de registo), o *behaviour* acaba e invoca o *SetupGame()*, e a partir desse momento, não é possível adicionar mais nenhuma equipa ou soldado ao jogo.

- ***SetupGame()***

Um *FiniteStateMachineBehaviour*, composto por 2 estados, o primeiro, *StartingPositions()*, responsável por determinar as posições iniciais dos soldados, quer de forma aleatória quer com posições pré-definidas, para realização de testes, e o segundo, *RandomTeam()*, que escolhe aleatoriamente uma das equipas para começar a jogar. Quando termina, invoca os *behaviours IniciateNextTurn()* e *PrepareNextTurn()*.

- ***IniciateNextTurn()***

Este é um *FiniteStateMachineBehaviour*, com um total de 4 estados.

O primeiro, *SearchSurrounded()*, é responsável por verificar se existem soldados cercados. Este processo é feito calculando, para cada direção, uma variável booleana (falso se está livre, verdadeiro se está ocupado ou está no limite do espaço de jogo), e realiza a conjunção de todas. Assim, se o resultado da conjunção é verdade, significa que o soldado está cercado, sendo-lhe enviada uma mensagem com a *performative order_kill*, e dando-se a atualização do tabuleiro e das variáveis responsáveis pela gestão das equipas e soldados do próprio moderador. De seguida, verifica se o número de soldados de uma determinada equipa é menor do que 2 e, caso seja, procede à eliminação do último soldado, caso ainda esteja vivo, e remove a equipa do dicionário *registered_teams_dic*.

O segundo estado, *VerifyEnd()*, verifica se o número de equipas ainda em jogo é menor do que 2 e, caso seja, o jogo dá-se por terminado, ganhando a última equipa em jogo, ou empatando, caso não haja nenhuma, e procede a realizar o *stop()* do moderador.

Caso o jogo não tenha acabado, procede para o terceiro estado, *SendInfo()*, que envia, numa mensagem com a *performative inform*, para cada soldado todos os elementos

numa visão 7 por 7 em sua volta, incluindo amigos, inimigos, e o próprio soldado, para ele atualizar as suas coordenadas depois de um movimento.

Por último, o estado *Play()* envia uma mensagem com a *performative your_turn* a todos os soldados da equipa guardada na variável *playing_team*, para eles jogarem o seu turno, e dá-se o término deste *behaviour*.

- ***PrepareNextTurn()***

Das propriedades ACID (Atomicidade, Consistência, Isolamento e Durabilidade), verificamos que os movimentos dos soldados têm de apresentar algumas destas propriedades, nomeadamente a atomicidade, a consistência, e a durabilidade.

Assim, este *CyclicBehaviour*, depois de receber, através de mensagens com a *performative ask_move*, os movimentos pretendidos de todos os soldados vivos da equipa em jogo, verifica se estes são consistentes (ou seja, se eles querem para ir para um lugar livre no tabuleiro).

O problema principal surge quando um soldado quer ir para o lugar de um amigo que se vai mover nessa jogada. Se o seu amigo tiver prioridade, o lugar fica livre, e logo não há problema. Porém, se o próprio tem prioridade, quando ele se tentar mover, o moderador verifica que o lugar está ocupado e pode marcar, erroneamente, o conjunto de movimentos como sendo inconsistentes, e, portanto, não realizar nenhum. A prioridade de um soldado é definida pela ordem em que ele está registado na lista da sua equipa.

Desta forma, para dar a volta a este problema, e garantir as propriedades acima referidas, após receber todos os pedidos, o *behaviour* cria um tabuleiro temporário apenas com os inimigos da equipa que está a jogar, e com base nas coordenadas dos soldados e nos seus movimentos, executa-os um a um no tabuleiro temporário, e se nenhum dos soldados se sobrepuser a outro, significa que os movimentos são consistentes e, portanto, podem ser executados todos no tabuleiro principal. Caso contrário, nenhum deles é executado e passa a jogada à frente (perda de turno por má gestão do comandante), garantindo assim a atomicidade.

Porém, a alteração do tabuleiro principal pode ter problemas quanto à durabilidade dos movimentos, pois, para o alterar, é necessário não só colocar o soldado na sua nova

posição, mas também removê-lo da sua posição antiga. Por exemplo, na situação referida anteriormente, se o soldado referido tiver prioridade e for para a posição do amigo, ao alterar, posteriormente, a posição do amigo, se não tivermos cuidado, ao apagar a sua posição anterior do segundo, podemos apagar o primeiro soldado que se moveu para a posição do segundo, e assim ele desaparece do tabuleiro. Porém, esta situação revolve-se apagando a posição anterior apenas se esta está a ser ocupada com o soldado que estamos a mover. Esta verificação é feita através do JID do soldado em movimento.

Depois dos movimentos serem (ou não) executados, este *behaviour* acaba por mudar a equipa que está a jogar, adicionando 1 à variável *playing_team* ou igualando-a a 0 se esta é igual ao número de equipas a jogar, repetindo este processo iterativamente até chegar a uma equipa que tenha soldados vivos, e invoca o *behaviour IniciateNextTurn()* para dar início ao próximo turno.

Desta forma, o *CyclicBehaviour PrepareNextTurn()* está sempre funcional depois de ser invocado pelo *SetupGame()*, e o *FiniteStateMachineBehaviour IniciateNextTurn()* é invocado 1 vez no início pelo *SetupGame()* e tantas vezes quantos turnos adicionais existirem pelo *PrepareNextTurn()*.

Comandante

O comandante é o agente responsável pelo comando dos soldados e da estratégia a ser utilizada pela sua equipa. Existirão tantos Comandantes como equipas em jogo.

Tendo isto em conta, precisa de um conjunto de parâmetros internos para a realização destas tarefas, como a sua informação (da classe *InfoCommander*, com o seu JID e a sua equipa), as dimensões do jogo (x_*max* e y_*max*, inteiros), um contador de mensagens (*msg_counter*, inteiro), para percecionar quando é que recebe todas as mensagens dos seus soldados vivos, dois dicionários para os seus soldados vivos e os inimigos que eles veem (*soldiers_alive* e *enemies*, respetivamente, sendo as *keys* os JIDs deles, e os *values* o *InfoSoldado* respetivo), um dicionário para as ordens que ele quer dar a cada soldado (*orders*, com a *key* sendo o JID do soldado vivo, e o *value* o tipo de movimento). Na etapa final, foi-lhe adicionado também um dicionário responsável por manter a memória dos percursos (dinâmicos) que cada soldado tem de fazer no momento

de procura de inimigos (*search_routes*, com a *key* sendo o JID do soldado e o *value* uma lista das coordenadas onde ele tem de passar).

No final, este agente ficou com 7 *behaviours*, 3 deles essenciais (*CommanderRecieve()*, *Headquarters()* e *SendOrders()*), comuns a todas as estratégias, e os outros 4 mutuamente exclusivos (*NoMove()*, *RandomMove()*, *ScoreAttack()* e *ScoreStrategy()*), cada um referente a uma das estratégias desenvolvidas.

- ***CommanderRecieve()***

Este *CyclicBehaviour*, iniciado e terminado a par do agente, é responsável por receber e tratar as mensagens direcionadas ao mesmo.

Se a mensagem apresentar a *performative died*, é porque o soldado que a enviou morreu, e logo procede à sua remoção do dicionário *soldiers_alive*.

Caso a *performative* seja *soldier_intel*, significa que é o turno dele de jogar. As mensagens trazem consigo a visão de todos os seus soldados, pelo que as utiliza para atualizar as suas variáveis referentes aos seus soldados e aos inimigos que eles veem, esquecendo qualquer inimigo que deixa de estar em visão. Tendo recebido todas as mensagens dos soldados vivos, procede à invocação do behaviour *Headquarters()*.

- ***Headquarters()***

Este *OneShotBehaviour* é responsável por escolher a estratégia que irá calcular os movimentos de todos os soldados. Este foi criado para permitir que as equipas do jogo seguissem estratégias diferentes, para melhor demonstrar o funcionamento das estratégias e comparar os desempenhos delas.

Portanto, limita-se a invocar um dos 4 *behaviours* que implementam as estratégias referidas anteriormente (*NoMove()*, *RandomMove()*, *ScoreAttack()* e *ScoreStrategy()*). Todos estes 4 *OneShotBehaviours* têm dois resultados em comum: a atualização do dicionário *orders* e invocação do *behaviour SendOrders()*.

- ***SendOrders()***

Este *OneShotBehaviour* envia para cada soldado presente nas *keys* do dicionário *orders* uma mensagem com a *performative movement_order* e com a informação do seu movimento, “stay”, “up”, “down”, “left” ou “right”.

Desenvolvimento das estratégias e algoritmos

- ***NoMove()***

Como referido anteriormente, a primeira fase de desenvolvimento é verificar, através do *display*, que os agentes e o jogo estão a inicializar corretamente.

Para tal, criou-se o *OneShotBehaviour NoMove()* do comandante, que faz com que os seus soldados estejam todos parados, executando o movimento "stay" em todos os turnos. O resultado está apresentado na Figura 1, ou seja, o moderador consegue atribuir coordenadas aleatórias a todos os soldados e é possível ver todos os soldados com as cores da sua equipa no *display*.

- ***RandomMove()***

De seguida, foi criado um algoritmo de teste cuja principal função era o teste de comunicação entre agentes, assim como a movimentação dos soldados e verificar o bom funcionamento dos *behaviours* do moderador.

Para este efeito, foi utilizado o *OneShotBehaviour RandomMove()*. O funcionamento deste *behaviour* necessita de alguns parâmetros básicos e, no entanto, essenciais para o que seria feito posteriormente. De forma semelhante ao moderador, para garantir a consistência dos movimentos selecionados, também é criada uma matriz temporária que inicialmente apenas contem os inimigos em visão. Nesse momento, um soldado de cada vez, calcula os movimentos possíveis que este pode realizar (movimentos para os quais ele não ocupa o lugar de ninguém, nem sai do espaço de jogo) e escolhe aleatoriamente um desses movimentos possíveis, adicionando esse soldado às coordenadas resultantes da matriz temporária, para que os seus amigos o possam evitar, e o dicionário *orders* é atualizado.

Assim, foi possível analisar possíveis erros nos processos do moderador relacionados com a "morte" dos soldados, bem como o fim do jogo. Claro que, para conseguir isto, as dimensões do espaço de jogo foram reduzidas para 10x10, de forma a aumentar a probabilidade dos soldados se chocarem e se cercarem.

- ***ScoreAttack()***

Numa terceira fase, começou-se por desenvolver um algoritmo muito simplista de ataque, que consistia em procurar pelos inimigos utilizando o *RandomMove()* e, uma vez encontrado um ou mais inimigos, escolhia, pelas distâncias de *Manhattan*, o que estivesse mais próximo de todos os soldados, sendo esse o *target*. De seguida, cria de novo a matriz temporária apenas com os inimigos, e soldado a soldado, compara as coordenadas do próprio com as do *target*, e desse modo escolhe um movimento possível que o aproxime do alvo. Ou seja, se o x do soldado for menor que o do *target*, então é interessante realizar o movimento "right", se for maior, é o "left", sendo o mesmo feito para o y com "up" e "down". Claro que, no caso apresentado na Figura 2, o soldado realçado a azul-claro ficava "preso" nos amigos. Como os seus x e y eram inferiores ao do *target*, então os movimentos interessantes eram "up" e "right", mas como já estavam ocupados, acabava por escolher o "stay".

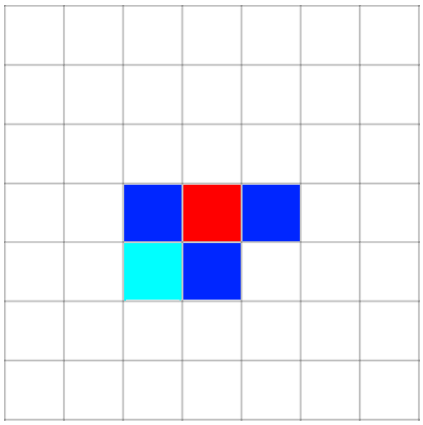

Figura 2 – Problema do primeiro algoritmo criado.

Para tentar contornar reste problema, optou-se por criar um sistema de *scores*, ou seja, atribuir um número avaliador a cada posição do espaço de jogo, com base na sua distância ao target e cada movimento do soldado seria escolhido com base no *score* da posição em que cada movimento o colocaria. O *score* máximo (determinado como sendo a soma das dimensões do espaço de jogos, ou seja, 200) ocorre nas posições adjacentes ao target e diminui em 1 unidade para cada posição adjacente a esta, o que resulta numa matriz semelhante à demonstrada na Figura 3.

195	196	197	198	197	196	195
196	197	198	199	198	197	196
197	198	199	200	199	198	197
198	199	200	t	200	199	198
197	198	199	200	199	198	197
196	197	198	199	198	197	196
195	196	197	198	197	196	195

Figura 3 – *Scores* em torno do *target*.

De forma a não ter de calcular 10.000 *scores* (100x100) em cada turno, pode-se calcular apenas os *scores* dos 5 movimentos que o soldado pode fazer, subtraindo ao *score* máximo a distância de *Manhattan* da posição resultante do movimento até ao *target*. Porém, como demonstrado na Figura 4, isto ainda não resolve o problema, pois como as posições com o *score* máximo estão ocupadas, o maior *score* restante (não ocupado) é o *score* da posição onde o soldado está, pelo que não se moveria.

195	196	197	198	197	196	195
196	197	198	199	198	197	196
197	198	199	200	199	198	197
198	199	200	t	200	199	198
197	198	199	200	199	198	197
196	197	198	199	198	197	196
195	196	197	198	197	196	195

Figura 4 – Permanência do problema com o algoritmo de *scores*.

Assim, seria do interesse que o *score* "contornasse" os soldados que estão a fazer de obstáculo até ao *target*. Para tal, criou-se um algoritmo, baseado no algoritmo de

procura primeiro em largura, na medida em que apresenta uma lista de espera dos nós não explorados, e que começa no nó inicial, neste caso, a posição do *target*, e os nós posteriores são adicionados no final da lista. Assim, como o primeiro nó se trata do *target*, adiciona o *score* máximo às posições livres adjacentes ao mesmo, adiciona essas posições ao fim da lista de espera e remove a posição do *target*. E enquanto a lista não estiver vazia, pega no primeiro elemento, calcula o *score* dos seus vizinhos livres, que ainda não têm *score*, como sendo o seu próprio menos uma unidade e adiciona-os ao fim da fila.

Para não calcular 10.000 *scores*, limita-se a procura ao campo de visão coletivo dos soldados e, portanto, caso os soldados estejam separados de forma que as suas visões se toquem, mas não se sobreponham, o número máximo de *scores* que calculará é de cerca de 250, uma diminuição significativa. Caso exista um ou mais soldados cuja visão não se encontra neste conjunto, é utilizado o método anterior para estimar os *scores* dos seus movimentos e aproximá-lo do target e estarem todos os soldados envolvidos numa visão coletiva única.

Implementando este novo método, o problema acima referido é resolvido, como demonstrado na Figura 5. Idealmente, o soldado acima do realçado mover-se-ia para a posição acima do *target*, o que pode, ou não acontecer. Se o soldado realçado tiver prioridade no momento da escolha do movimento, ele irá para a posição do soldado acima, e quando o segundo for escolher o seu movimento, o melhor é seguir para a posição acima do *target*. Caso o soldado acima tenha prioridade, o soldado realçado terá de o contornar.

195	196	197	198	197	196	195
196	197	198	199	198	197	196
197	198	199	200	199	198	197
196	197	a	t	a	197	196
195	196	195	a	195	196	195
194	195	194	193	194	195	194
193	194	193	192	193	194	193

Figura 5 – Novo algoritmo de scores.

- ***ScoreStrategy()***

Para completar o algoritmo anterior, falta desenvolver uma procura também elaborada e não apenas aleatória. Idealmente, quando os soldados não tivessem nenhum inimigo em visão, eles organizar-se-iam numa formação que lhes permitisse atingir dois objetivos: estarem perto uns dos outros, para atacar de forma rápida os inimigos que encontrarem e estarem espaçados o suficiente para "varrer" todo o espaço de jogo no menor número de turnos possível.

Desta forma, definimos que a formação a explorar seria uma formação em linha reta, em que os soldados se organizam paralelos a um dos eixos e encostado a outro (portanto, começam num dos cantos do espaço de jogo), com uma distância de exatamente 6 posições entre eles e de 3 posições com os limites do espaço de jogo, para que os seus campos de visão se toquem, mas não se sobreponham, nem ultrapassem o espaço. O canto a partir do qual a procura começará a ser feita é escolhido calculando o canto do tabuleiro mais próximo dos soldados vivos. Quando estiverem todos em posição, avançam de forma perpendicular à sua formação, até chegar ao outro limite do espaço de jogo. Uma vez que a visão deles atinge esse limite, a formação avança de forma paralela para que, quando voltarem a "varrer" em sentido oposto, não o façam na "banda" em que acabaram de procurar. Sempre que chegarem ao final do percurso ou encontrarem e eliminarem um inimigo, o processo é recomeçado do zero.

Como se vê na Figura 6, a equipa azul decidiu procurar na horizontal, começando no canto superior direito. Desta forma, se não forem interrompidos, irão descer até o seu y ser 3, depois vão se mover para a esquerda, voltar a subir até o y ser 96, voltar a mover para a esquerda, até ao x do elemento mais à esquerda ser 3, e voltar a descer até ao y de 3, onde recomeçarão. A equipa vermelha funcionaria de forma semelhante, mas em vez de ser na horizontal, seria na vertical e começando no canto inferior esquerdo.

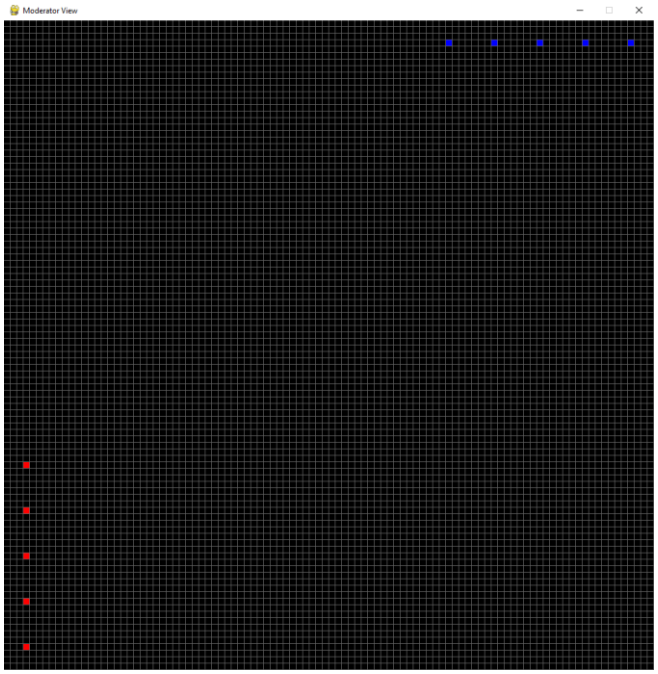

Figura 6 – Início do algoritmo de procura.

Para o funcionamento deste algoritmo, é utilizado o método de *score* desenvolvido anteriormente, mas com a alteração no modo com que se calcula o *target*, que desta vez são as posições perto nos limites do espaço de jogo. Aqui, foi necessário melhorar o algoritmo de estimativa do *score*, pois no caso de ataque, todos têm o mesmo *target*, logo, convergem no mesmo ponto, mas no de procura, com *targets* diferentes, os soldados podem ficar presos uns nos outros, como demonstrado na Figura 7, em que o soldado a azul-escuro está no seu lugar da formação, mas o azul-claro, que tem de ir até à posição a verde, fora da visão conjunta da equipa, permanece no lugar, porque como a posição com maior *score* estimado está ocupada, o movimento que gera a posição com o segundo maior *score* é o "stay".

195	196	197	198	197	196	195	194	193	192	191	190	189	188
196	197	198	199	198	197	196	195	194	193	192	191	190	189
197	198	199	200	199	198	197	196	195	194	193	192	191	190
198	199	200	201	200	199	198	197	196	195	194	193	192	191
197	198	199	200	199	198	197	196	195	194	193	192	191	190
196	197	198	199	198	197	196	195	194	193	192	191	190	189
195	196	197	198	197	196	195	194	193	192	191	190	189	188

Figura 7 – Problema com o algoritmo de estimativa do *score*.

Desta forma, quando o *behaviour* se depara numa situação destas, soma 3 às 2 posições laterais, e 2 à posição posterior (para que o soldado só volte para "trás" no caso de ambas as posições laterais estarem ocupadas), como demonstrado na Figura 8.

195	196	197	198	197	196	195	194	193	192	191	190	189	188
196	197	198	199	198	197	196	195	194	193	192	191	190	189
197	198	199	200	199	198	197	196	195	194	193	195	191	190
198	199	200	201	200	199	198	197	196	195		193	194	191
197	198	199	200	199	198	197	196	195	194	193	195	191	190
196	197	198	199	198	197	196	195	194	193	192	191	190	189
195	196	197	198	197	196	195	194	193	192	191	190	189	188

Figura 8 – Primeira melhoria na estimativa do *score*.

Assim, o soldado avançará ou para cima ou para baixo, porém, depara-se com um problema novo, na jogada a seguir, poderá voltar para a posição onde estava, como demonstrado na Figura 9.

195	196	197	198	197	196	195	194	193	192	191	190	189	188
196	197	198	199	198	197	196	195	194	193	192	191	190	189
197	198	199	200	199	198	197	196	195	194	193	192	191	190
198	199	200	201	200	199	198	197	196	195		193	192	191
197	198	199	200	199	198	197	196	195	194	193	192	191	190
196	197	198	199	198	197	196	195	194	193	192	191	190	189
195	196	197	198	197	196	195	194	193	192	191	190	189	188

Figura 9 – *Score* das posições depois do movimento "down".

Para resolver esta situação, atribui-se, aos *scores* das posições adjacentes, o *score* da posição seguinte, como demonstrado na Figura 10, o que permite que o soldado contorne o "obstáculo".

195	196	197	198	197	196	195	194	193	192	191	190	189	188
196	197	198	199	198	197	196	195	194	193	192	191	190	189
197	198	199	200	199	198	197	196	195	194	193	192	191	190
198	199	200	201	200	199	198	197	196	195		192	192	191
197	198	199	200	199	198	197	196	195	194	194	192	190	190
196	197	198	199	198	197	196	195	194	193	192	190	190	189
195	196	197	198	197	196	195	194	193	192	191	190	189	188

Figura 10 – Segunda melhoria na estimativa do *score*.

Note-se que este algoritmo de procura é interrompido sempre que um inimigo é detetado por algum dos soldados e é retomado se não existir nenhum inimigo no campo de visão dos soldados após um ataque.

Conclusão

O trabalho desenvolvido permitiu consolidar conhecimentos no que toca ao desenvolvimento e implementação de sistemas multiagentes, utilizando agentes inteligentes.

Foi possível a criação de um jogo 5v5 num quadro de 100x100, em que os diversos agentes de cada equipa são capazes de percecionar e interpretar o ambiente que os rodeia, e tomar decisões inteligentes consoante a situação, de forma cooperativa e competitiva.

No entanto, apesar do funcionamento correto do algoritmo, as soluções encontradas muito possivelmente não serão as soluções ótimas, existindo espaço para melhoria. Uma possibilidade seria a implementação de mais formações, tendo em conta o que se pretende fazer, por exemplo, quando se inicia uma tentativa de captura, uma formação em “w” ou “v” para tentar afunilar inimigos para posições específicas.

Outra melhoria poderia ser a implementação de algoritmos que tivessem em conta possíveis jogadas posteriores do adversário, tentando prevê-las e assim aumentar as suas chances de vitória.

Seria ainda de salientar que os agentes não têm memória dos outros, ou seja, depois de um inimigo desaparecer do seu campo de visão, estes esquecem-se da sua posição, não sendo capazes de continuar uma perseguição depois disto acontecer.

Assim sendo, apesar de existirem áreas aonde ainda poderiam ser efetuadas melhorias, o programa consegue cumprir as suas funções e reagir adequadamente quando confrontado com as diversas situações que podem ocorrer no jogo, sendo que as melhorias acima propostas apenas lhe serviriam para aumentar a eficiência e eficácia.

Anexos

Anexo 1

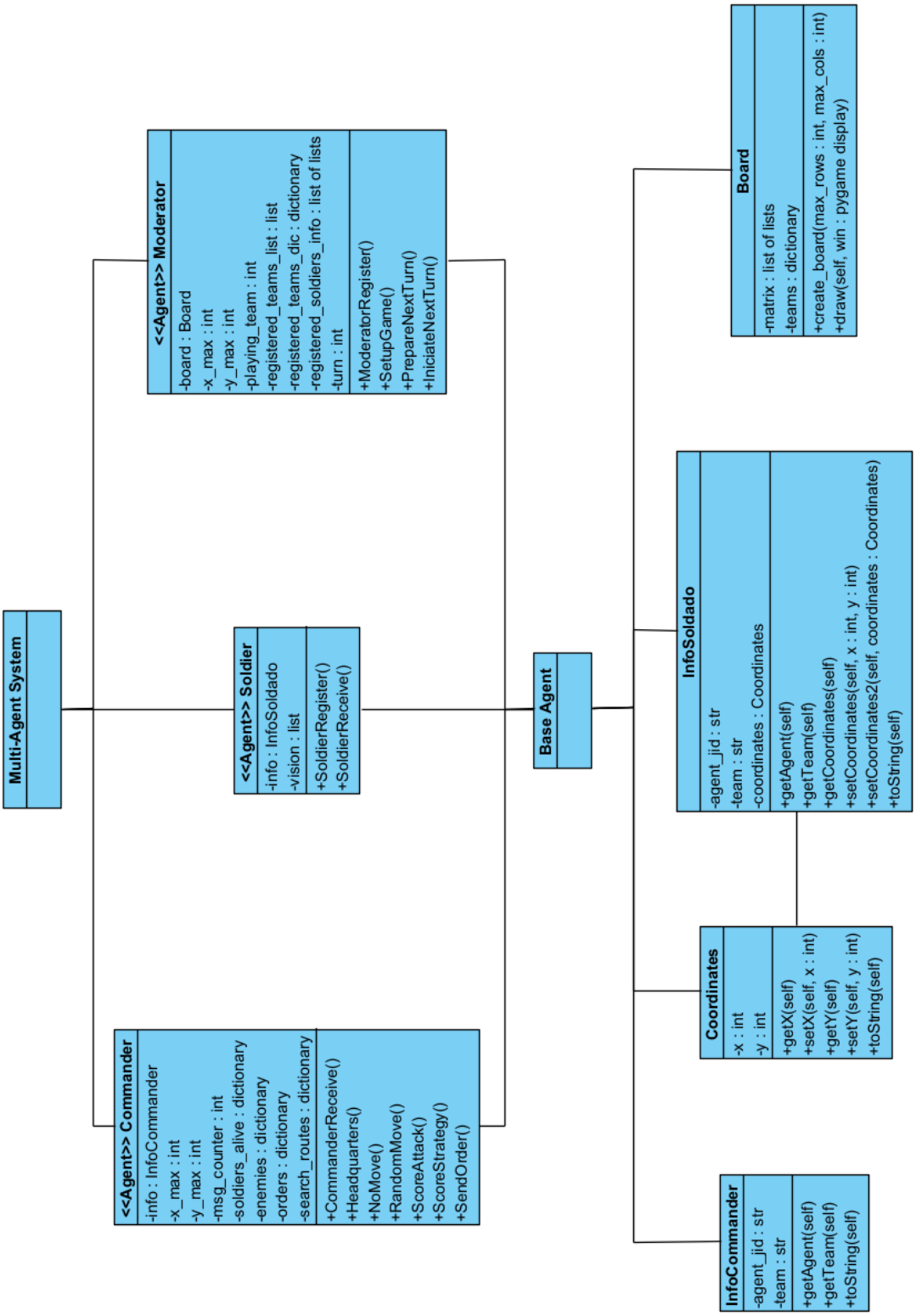

Figura 1 – Diagrama de Classes.

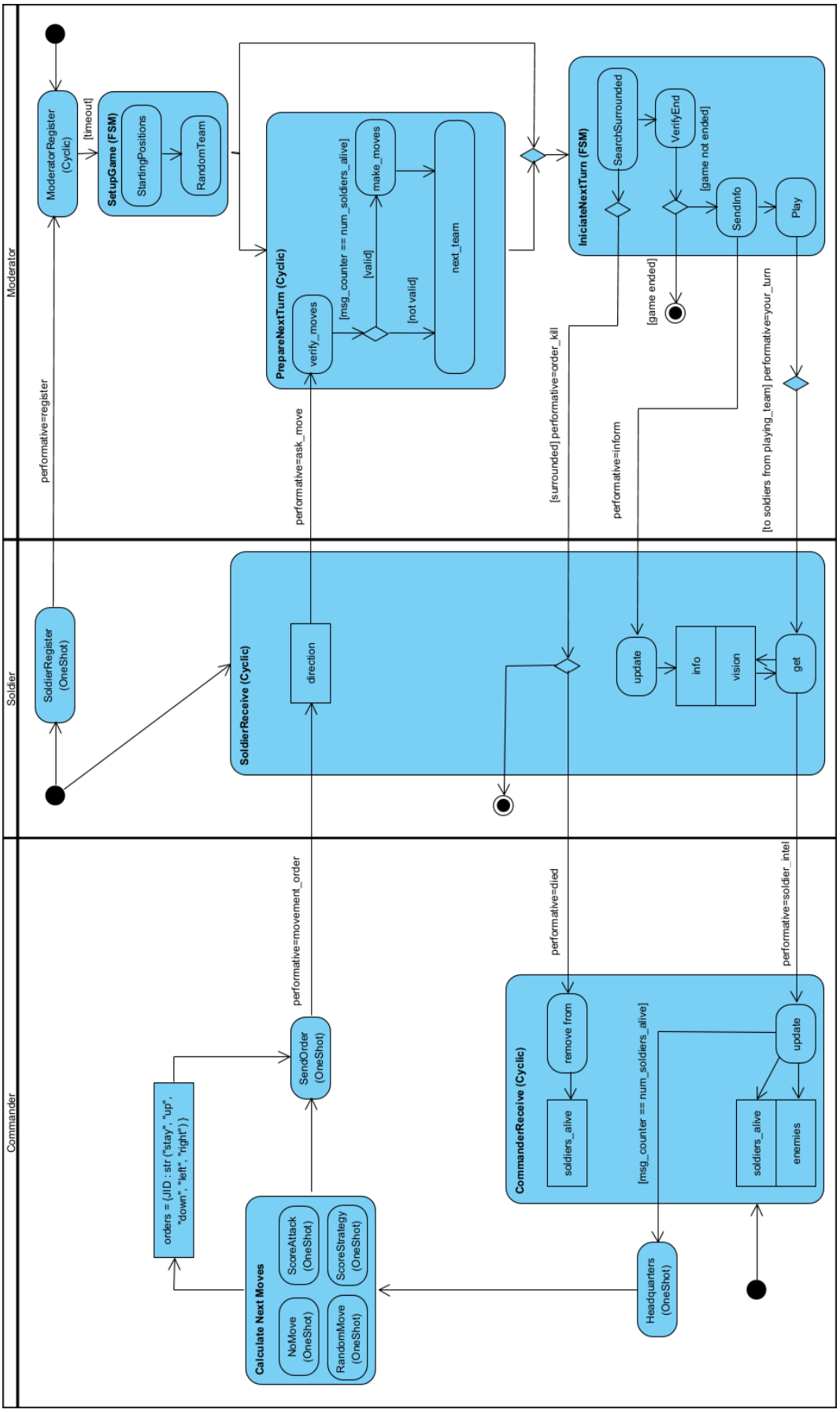

Figura 2 – Diagrama de Atividades.