Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores

[Relatório Final - IASA]

47202 : Manuel Henriques (47202@alunos.isel.pt)

Relatório para a Unidade Curricular de Inteligência Artificial para Sistemas Autónomos da Licenciatura em Engenharia Informática e de Computadores

Professor : Engenheiro Paulo Vieira

26 – 06 – 2023

Índice

Lista de Figuras

1

Introdução

Neste relatório, realizado no âmbito da cadeira de Inteligência Artificial para Sistemas Autónomos, será apresentado o projeto final desenvolvido ao longo do semestre. O projeto visa o estudo e implementação, em linguagem Python, de diversos algoritmos e abordagens referentes à criação de um agente autónomo. Como base para o desenvolvimento deste este projeto, utilizamos um ambiente, criado através bibliotecas Pygame e SAE(fornecida pelos docentes), constituído pelos seguintes elementos:

- **Agente**: componente central do ambiente. Ele é responsável por aplicar os algoritmos e abordagens de inteligência artificial estudados durante as aulas. O agente tem a capacidade de perceber e interagir com o ambiente, tomar decisões com base nessas perceções e executar ações para se movimentar pelo ambiente.
- **Alvo**: O alvo é o estado que o agente deseja atingir. O alvo serve como uma referência para o agente direcionar as suas ações e tomar decisões visando alcançá-lo.
- **Obstáculo**: são elementos que visão representar barreiras físicas que impedem a movimentação do agente. Criam restrições e impedem o progresso do agente em determinadas posições do ambiente.

O projeto ficou separado em dois módulos distintos.

No primeiro, iasa_jogo, tivemos a oportunidade de explorar e compreender conceitos fundamentais, como modelos dinâmicos e agentes reativos. Essa etapa inicial foi de extrema importância, pois estabeleceu as bases para as fases subsequentes do projeto.

Foi também nesta fase que nos familiarizámos com esquemas UML (Unified Modeling Language), que se tornariam componentes essenciais para interpretar as diferentes fases do projeto.

Durante o módulo iasa_agente, aprofundámo-nos na exploração de algoritmos de busca, que nos permitiram encontrar caminhos ótimos em espaços de estado complexos. Aprendemos a aplicar heurísticas adequadas para guiar a busca e a lidar com a complexidade de problemas de grande escala. Além disso, fomos introduzidos a processos de decisão, nos quais exploramos conceitos como a teoria da decisão e aprendizado por reforço.

2

Enquadramento Teórico

Nesta fase do relatório, é pretendido apresentar todos os fundamentos teóricos e conceituais estudados e utilizados como base para o desenvolvimento do trabalho.

2.1 UML

Devido à alta complexidade dos sistemas relacionados à inteligência artificial, é extremamente difícil definir as características de um sistema de forma objetiva, simples e universal. A fim de sermos capazes de representar e compreender os sistemas que usaríamos, começamos por estudar UML.

UML, ou *Unified Modeling Language* , é uma linguagem de modelação de sistemas que visa a representação sucinta de estruturas, interações e funcionalidades dos mesmos, sendo amplamente usada para a modelação de sistemas de *software*.

Uma vez que a linguagem UML não está diretamente relacionada à componente teórica, no âmbito de inteligência artificial, utilizada neste projeto, não me irei adentrar nas suas características, no entanto, é importante mencionar que foi um componente essencial para compreensão e planeamento de todas as fases do projeto.

2.2 Arquitetura de agentes reativos

2.2.1 Bases da arquitetura

A arquitetura de agentes reativos visa a criação de um agente autónomo cujo comportamento é gerado de forma reativa, ou seja, as ações tomadas pelo agente resultam das interações que este tem com o ambiente em que está inserido, nomeadamente por via de estímulos. Assim, chegamos a uma arquitetura que se baseia no seguinte:

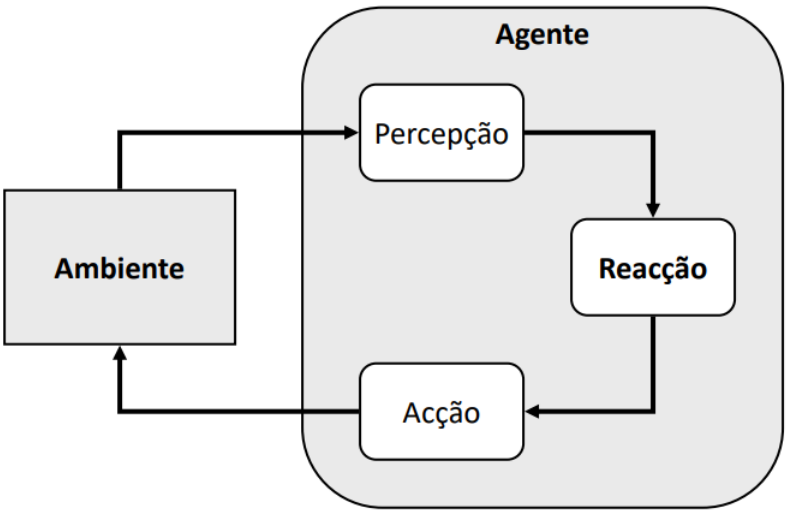

Figura 2.1: Arquitetura de agentes reativos

Neste ciclo, nomeado de ciclo perceção-reação-ação, agente é capaz de recolher informações do ambiente em que está inserido e, através dessa informação, provocar estímulos que permitem perceber o seu estado atual. É através do processamento desses estímulos que o agente escolhe a reação prioritária, que por sua vez resulta numa ação de resposta.

É importante notar que, em arquiteturas mais simples, o agente não possui quaisquer representações internas do estado do ambiente. Isto significa que a seleção das respostas é feita exclusivamente com base nas perceções recebidas pelo agente. Embora esta característica torne a solução consideravelmente mais simples, tanto numa perspetiva de complexidade de arquitetura como de carga computacional e utilização de recursos, também adiciona algumas limitações. Nomeadamente, esta arquitetura é incapaz de planeamento a longo prazo, está suscetível a repetir padrões redundantes(repetir ações ou comportamento cíclico) além de puder ter dificuldade em lidar com situações que requerem uma coordenação mais avançada.

2.2.2 Comportamentos

Um comportamento corresponde a um encapsulamento e várias reações, relacionadas entre si, num único modulo comportamental. Cada comportamento está associado a um conjunto de padrões de comportamento, que estabelecem uma relação direta entre os estímulos recebidos pelo agente e as ações a serem tomadas em resposta a esses estímulos.

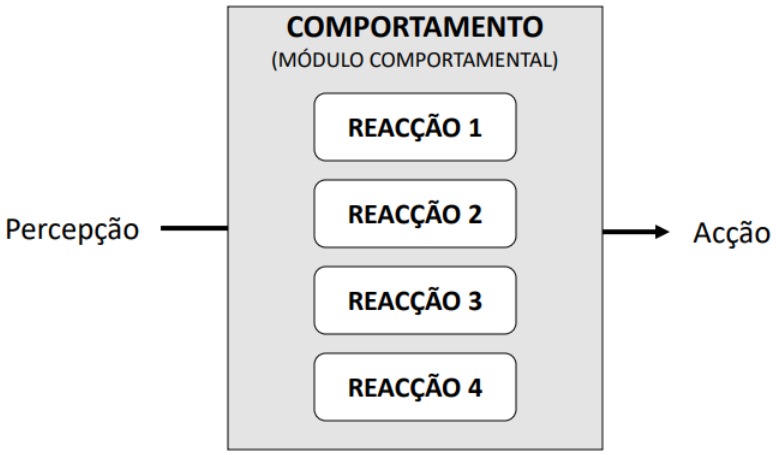

Figura 2.2: Comportamento

Com as reações agrupadas em comportamentos, falta agora compreender como são selecionadas as ações em função do estímulo recebido.

Para isso, existem três possíveis mecanismos de seleção:

- **Execução paralela**: todas as ações do comportamento são executadas em paralelo.
- **Combinação**: as ações possuem um nível de prioridade, sendo a ação mais prioritária aquela executada.
- **Precedências**: as várias ações são combinadas numa única ação composta, que é executada.

Além de reações, comportamentos são também capazes de encapsular outros comportamentos, sendo assim classificados como **comportamentos compostos**. No entanto, a capacidade de agrupar vários comportamentos diferentes exige também que exista um mecanismo que, dado o estímulo, seja capaz de selecionar qual dos comportamentos deve ser utilizado para determinar a ação. Os mecanismos utilizados no âmbito do projeto foram os seguintes:

- **Hierarquia**: Os vários comportamentos são ordenados hierarquicamente de modo que, uma ação que resulte de um comportamento hierarquicamente superior seja capaz de suprimir as ações dos comportamentos abaixo.
- **Prioridade**: As ações são selecionadas de acordo com um nível de prioridade que pode variar.

2.2.3 Arquitetura com memória

Como visto anteriormente, uma arquitetura que não é capaz de manter em memória uma representação interna de estado levanta diversos problemas que podem condicionar muito o comportamento do nosso agente. Assim, para que o nosso agente seja capaz de comportamentos eficazes e efetivos, torna-se essencial a manutenção de estado.

Ao adicionar um componente de memória à arquitetura, as ações do agente são influenciadas não apenas pela perceção atual, mas também pelas informações armazenadas na memória sobre perceções anteriores. Isso permite ao agente tomar decisões mais complexas, informadas e adaptativas.

2.3 Procura em espaços de estados

Uma procura em espaço de estados é um método de exploração sistemática que, fornecido um conjunto de estados, um estado inicial e um estado final, percorre as possíveis transições entre os estados até encontrar uma sequência que leve do estado inicial ao estado final. Essa sequência de estados representa uma solução para o problema em questão.

Este mecanismo de procura permite um elevado nível de abstração do problema, uma vez que os **estados** podem corresponder a uma qualquer configuração possível na resolução. A transição de estados é feita por via de **operadores**, que quando aplicado a um estado é capaz de gerar um novo estado. O problema é solucionado assim que um estado sofre uma transição, por via de um operador, para o estado final, também chamado de objetivo.

2.3.1 Mecanismo de procura

Antes de sermos capazes de entender como funciona o mecanismo de procura, precisamos de entender o que é a **fronteira**. No processo de procura, devido à complexidade

dos modelos, existe a necessidade de guardar os estados que resultaram de outras expansões e, que assim, ainda precisam de ser explorados. Para esse efeito, foi criada a fronteira de exploração, uma lista que representa os estados prontos para serem expandidos em etapas posteriores da busca.

O mecanismo de procura, independentemente dos métodos de procura utilizados, segue um padrão geral. Inicialmente, verifica-se se o estado atual corresponde ao estado final desejado. Se for o caso, uma solução é encontrada. No entanto, se o estado atual não for o estado final, ocorre a etapa de expansão. Durante a expansão, todos os operadores disponíveis são aplicados ao estado atual, gerando novos estados. Esses novos estados resultantes das expansões são adicionados à fronteira de exploração, de onde mais tarde serão retirados e processados seguindo este mesmo padrão. Este ciclo repete-se até que a solução seja encontrada ou não haja mais estados na fronteira para explorar, o que significa que não existe solução possível.

Como forma auxiliar a compreensão da exploração, pudemos associar o resultado do mecanismo de procura a um grafo em árvore, onde os nós representam os estados explorados e as arestas representam as transições entre esses estados.

Com o padrão de procura explicado, é relevante ressaltar que a busca em espaços de estados pode ser implementada utilizando diferentes métodos de procura.

2.3.2 Métodos de procura

Os métodos de procura corresponde aos algoritmos de utilizados na exploração do espaço de estados. A escolha do método de procura tem impacto significativo em vários aspetos da exploração, nomeadamente na forma como o mecanismo utiliza a fronteira, especialmente em relação aos nós expandidos que não devem ser guardados e na ordem em que os nós são removidos da fronteira de exploração.

Os métodos de procura podem ser divididos em dois tipos:

- **Métodos de procura informada**: Procura selectiva que tira partido de conhecimento do domínio do problema para ordenar a fronteira de exploração
- **Métodos de procura não informada**: Procura exaustiva e que não tira partido de conhecimento do domínio do problema para ordenar a fronteira de exploração

Além disso, um método de procura é considerado **completo** caso garante que, independentemente do caso, se existir uma solução, vai conseguir encontrá-la. Por outro lado, o método é chamado de **oprimo** caso garanta que, na existência de várias soluções possíveis, a encontrada seja sempre a melhor.

2.3.2.1 Procura em profundidade

A procura em profundidade consiste em explorar o máximo possível numa direção antes de retroceder. Isto é possível, pois, seguindo este método, os nós removidos da fronteira são sempre os mais recentes, o que faz com que a busca se aprofunde continuamente num ramo específico do espaço de estados até atingir o objetivo ou um nó que não consiga expandir mais, antes de retornar e explorar outros ramos.

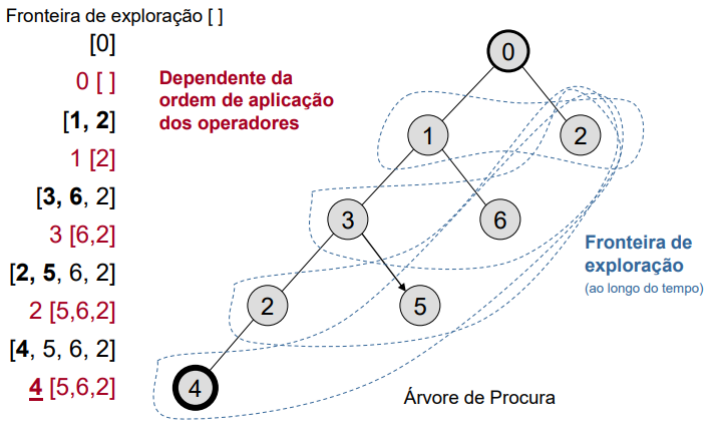

Figura 2.3: Procura em profundidade

É um método de procura não informado e não é óptimo nem completo. Devido à forma como faz a exploração, está suscetível a ficar preso em loops infinitos.

2.3.2.2 Procura em largura

A procura em largura apresenta uma abordagem oposta à da procura em profundidade, uma vez que explora os nós em níveis crescentes de profundidade. Isto significa que, ao explorar os nós em largura, a procura em largura garante que todos os nós de uma determinada profundidade sejam visitados antes de prosseguir para o próximo nível de profundidade. Esta abordagem é possível porque os nós removidos da fronteira de exploração são sempre os mais antigos, ou seja, aqueles inseridos primeiro.

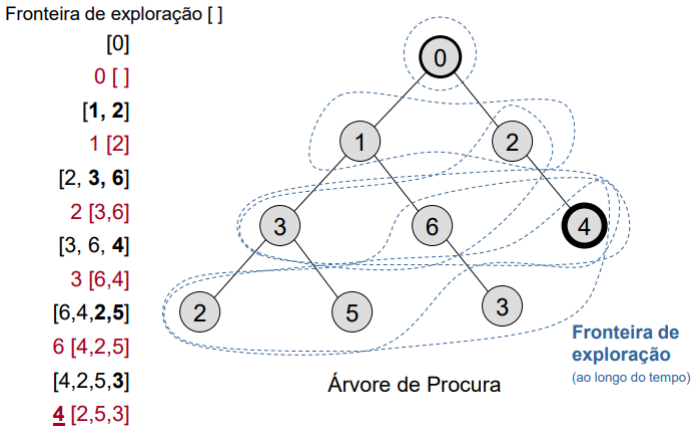

Figura 2.4: Procura em largura

É um método não informado, óptimo e completo.

2.3.2.3 Procura em profundidade limitada

A procura em profundidade limitada é uma variação da procura em profundidade que impõe uma restrição na profundidade máxima alcançada durante a exploração. Enquanto na procura em profundidade comum não existem limites de profundidade, na procura em profundidade limitada, os nós que atingem a profundidade máxima especificada não são mais explorados.

É um método não informado, não óptimo e não completo.

2.3.2.4 Procura em profundidade iterativa

A procura em profundidade iterativa é uma estratégia de exploração que realiza a iteração crescente do limite de profundidade enquanto realiza procuras de profundidade limitada. Inicialmente, executa uma procura de profundidade limitada com limite de 1 e, de seguida, repete o processo incrementando o limite até que chegue ao limite fornecido ou caso encontre promoção.

É um método não informado, óptimo e completo.

2.3.2.5 Procura melhor-primeiro

A procura de melhor-primeiro faz uso de uma função fornecida, **f(n)**, para avaliar cada nó, em função da sua **heurística**, ou seja, do custo do percurso até esse estado. A partir dessa função, os nós presentes na fronteira são ordenados por ordem crescente, onde serão retirados de forma ordenada, para exploração.

É um método informado, não óptimo e não completo.

2.4 Arquitetura de agentes deliberativos

A arquitetura de agentes deliberativos é uma extensão da arquitetura de agentes reativos com memória. Nessa arquitetura, há um foco significativo na capacidade de memória do agente para armazenar representações do ambiente, o que possibilita fornecer ao agente um mecanismo de raciocínio e tomada de decisão mais sofisticado.

Através da memória, o agente é capaz de relembrar informações passadas, que permitem que repita/evite ações passadas baseado nos seus resultados, analisar o estado atual do ambiente, de forma reagir a estímulos, e antecipar ações futuras por meio de simulações no seu modelo do ambiente em memória. O processamento destas diferentes dimensões temporais permite que o agente leve em consideração um contexto mais amplo ao tomar decisões, levando em conta informações anteriores e antecipando possíveis cenários futuros.

2.4.1 Processo de tomada de decisão

Para este projeto, o nosso agente deliberativo irá utilizar um raciocínio orientado para a ação, chamado de **raciocínio prático**. Este tipo de raciocínio pode ser divido em dois componentes essenciais: **deliberação** e **planeamento**. A deliberação é responsável para identificar os objetivos que o agente deve tentar alcançar, enquanto o planeamento consiste em desenvolver um plano que permita ao agente alcançar ditos objetivos.

No processo de tomada de decisão da arquitetura do agente deliberativo, existem várias etapas que devem ser seguidas para garantir que são tomadas as decisões corretas. Primeiramente, o agente observa o ambiente e atualiza o seu modelo do mundo armazenado em memória. Essa atualização permite que o agente esteja ciente do estado atual e das mudanças que ocorreram desde a última observação.

Em seguida, ocorre o processo de deliberação, onde o agente identifica os objetivos a serem alcançados e a posição atual no modelo do mundo.

É importante ressaltar que, uma vez que os objetivos sejam identificados, só ocorre um novo processo de deliberação caso se percebam mudanças no ambiente que não se encontrem no modelo do mundo armazenado em memória. Esta verificação evita um processo repetitivo de deliberação quando as condições do ambiente não foram alteradas, além de permitir que o agente de adapte a ambientes que se alteram de forma dinâmica.

Uma vez que os objetivos forem definidos, o agente passa para a etapa de planeamento, na qual são elaboradas as estratégias e ações necessárias para alcançar os objetivos estabelecidos.

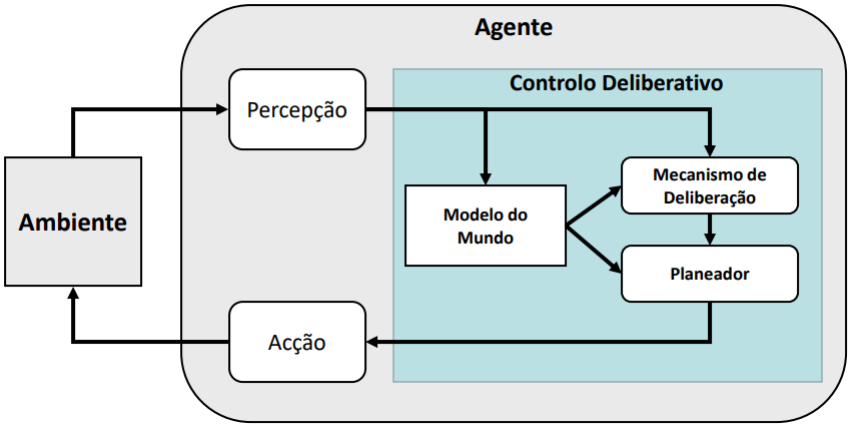

Figura 2.5: Arquitetura deliberativa

2.5 Processo de decisão sequencial

O processo de decisão sequencial é um mecanismo que permite ao agente tomar decisões de forma sequencial, considerando diferentes informações referentes ao ambiente, para determinar as melhores ações a serem tomadas. A partir dessas informações, é gerada uma política comportamental, ou seja, um conjunto de regras que orientam as ações a serem tomadas. Essa política determina as melhores movimentações que o agente pode fazer num determinado momento.

2.5.1 Recompensas

As recompensas são valores, associados a estados, que oferecem benefícios ou penalidades imediatas ao agente caso este os alcance. Essas recompensas podem ser positivas, representando benefícios ou ganhos para o agente, ou negativas, indicando penalidades ou perdas. Assim, dentro do contexto do processo de decisão sequencial, os estados com recompensas positivas são considerados desejáveis de alcançar e são os estados que o agente procurará encontrar.

No âmbito do projeto, só são usadas recompensas com valores positivos, que servirão para identificar os objetivos.

2.5.2 Utilidade

A utilidade é um valor, também associado a um estado, utilizado no processo de decisão de estados. O valor da utilidade é individual a cada estado e reflete o quão desejável um estado é para o agente.

No âmbito do projeto, uma vez que utilizamos o processo de decisão de Markov, utilidade é calculada em função da utilidade dos estados futuros. Isto implica que os estados com recompensas, bem como os estados adjacentes a eles, terão uma utilidade mais elevada. O valor da utilidade desses estados irá, por sua vez, ser passada para os estados sucessores, mas com um decréscimo calculado em função de **gamma**.

2.5.3 Política

A política, também chamada política comportamental, corresponde ao processo de seleção que, para cada estado do ambiente, escolhe a ação que deve ser tomada visando obter o melhor resultado. O método utilizado pode variar consoante seja uma política determinista ou não, mas de maneira geral, a política escolhe a ação que leva ao estado com maior grau de utilidade.

No projeto utilizamos uma política determinista, pois, ao aplicar uma ação sobre um estado, nunca poderá retornar mais do que um estado diferente. Portanto, uma vez que é determinista, a política irá inevitavelmente orientar o agente sempre para os estados com maior grau de utilidade.

3

Projeto realizado

3.1 Introdução

Neste módulo, iremos examinar detalhadamente cada uma das fases do projeto, com o objetivo de compreender as soluções desenvolvidas e como os conceitos teóricos foram aplicados para chegar a essas conclusões.

3.1.1 Parte 1

Iniciamos o projeto por realizar um pequeno jogo, como forma de nos familiarizarmos com o conceito de agente reativo e com a leitura de esquemas UML.

O conceito do jogo baseia-se na existência de um personagem(agente) que se encontra num ambiente onde existem diversos animais. O objetivo do agente é, por via de comandos inseridos pelo utilizador, procurar animais, aproximar-se e fotografar, enquanto o animal está imóvel.

O modelo do jogo, embora o agente não seja autónomo, segue uma arquitetura similar à arquitetura de agente reativo, na medida em que o agente obtém informações sobre o estado do ambiente, processa (sendo neste caso o processamento feito pelo jogador) e atua com base nessas mesmas informações.

3.1.2 Parte 2

Na segunda fase do projeto, realizamos a biblioteca *maqest* que será usada para complementar a implementação do jogo começado na fase anterior. Trata-se de uma biblioteca genérica de máquina de estados, constituída por três conceitos principais:

- **Transição**: Representa a transição do estado atual para um novo estado. É constituída pelo estado sucessor e ação que levou à transição.

- **Estado**: Representa um estado possível. É constituído por um nome e lista de transações possíveis.

- **Maquina Estados**: Máquina de estados, responsável por manter o estado atual e processar a mudança de estados.

De modo a manter a biblioteca o mais genérica possível, foram utilizados apenas os genéricos **EV**, que representa o tipo de evento, e **AC** que representa o tipo de ação.

A partir desta biblioteca, foi desenvolvido um módulo de controlo para o personagem do jogo. Esse módulo de controlo desempenha um papel fundamental na gestão do estado do agente, atualizando-o em função das ações tomadas.

3.1.3 Parte 3

A partir desta fase do projeto, deixamos de lado o jogo e começamos a trabalhar na implementação de uma biblioteca, chamada **ecr**, que dá suporte a implementações da arquitetura de agente reativo. Nesta fase, focámo-nos no mecanismo que, fornecendo uma perceção, é capaz de gerar uma reação e retorna a ação a ser tomada.

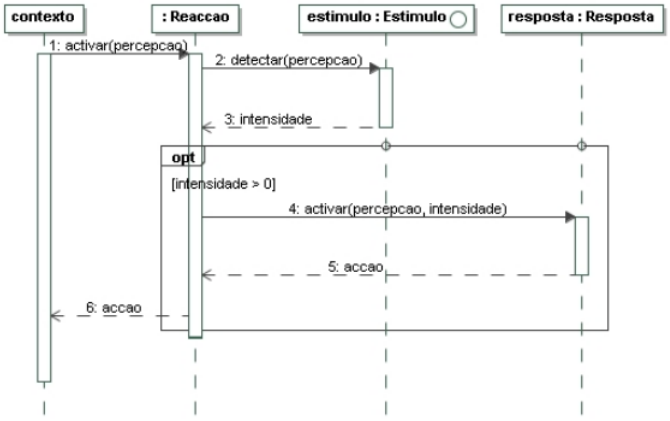

Figura 3.1: Reação

3.1.3.1 Reaccao

A classe Reaccao estende da classe abstrata Comportamento, também pertencente a esta biblioteca. O mecanismo funciona da seguinte forma: a perceção é utilizada para identificar o tipo de estímulo e a sua intensidade. Caso a intensidade não seja nula, significa que o agente deve reagir a esse estímulo, e assim ativar uma resposta, que por sua vez irá retornar a ação que deve ser tomada. A ação retornada é uma instância da classe Accao, presenta na biblioteca **sae** fornecida pelos docentes.

3.1.3.2 Comportamento

A classe abstrata Comportamento também é estendida pela classe ComportComp, que irá representar os comportamentos compostos. Além disso, existem outras classes, como a classe Estimulo, que possuem componentes abstratos. É importante entender que isto acontece porque a biblioteca não está a ser desenvolvida com o intuito de puder ser usada diretamente como uma arquitetura reativa, mas, em vez disso, fornecer a infraestrutura e suporte para implementações de terceiros.

3.1.4 Parte 4

A quarta fase tem o intuito de começar a criação do nosso controlo reativo que fará uso da biblioteca ecr (começada na última aula) e sae (fornecida pelos docentes). No entanto, primeiro precisámos de implementar o comportamento composto na biblioteca.

3.1.4.1 Comportamento composto

O comportamento composto é iniciado com o conjunto de sub-comportamentos que o constituem. Ao receber uma perceção, o comportamento ativa todos os sub-comportamentos e gera uma ação para cada um deles, guardando todas as não nulas numa lista. A ação retornada depende do mecanismo de seleção. A biblioteca oferece suporte a mecanismo de **prioridade** que retorna a ação da lista que possui o atributo *prioridade* com maior valor, e o mecanismo de **hierarquia**, que assume que os subcomportamentos já estão ordenados hierarquicamente e que, portanto devolve a primeira ação da lista.

3.1.4.2 Controlo reativo

De seguida, foi implementado o controlo reativo, responsável por gerir as ações do agente em função das perceções recebidas ao longo da execução e do comportamento estabelecido quando foi inicializado.

Juntamente com o controlo reativo, também implementámos o comportamento "Explorar". Este comportamento, independentemente da perceção recebida, irá retornar uma ação que fará o agente movimentar-se numa direção aleatória.

3.1.5 Parte 5

Com um controlo reativo já pronto e a funcionar, objetivo da fase 5 era aumentar o nível de competência do nosso agente, tornado o seu comportamento mais dinâmico e adaptativo por via de um comportamento composto apelidado de "Recolher".

O comportamento "Recolher"é um comportamento composto que utiliza um mecanismo de seleção hierárquica para determinar as ações do agente. Ele é composto por três subcomportamentos: "AproximarAlvo", "EvitarObst"e "Explorar", tendo a hierarquia esta mesma ordem.

O comportamento "AproximarAlvo", quando ativado, verifica se há algum alvo em qualquer uma das direções da perceção. Se um alvo for identificado, o comportamento gera uma ação que move o agente em direção a esse alvo. Uma vez que, por cada perceção, podem ser identificados alvos em múltiplas direções, este comportamento foi implementado também como um comportamento composto, mas com um mecanismo de seleção prioritário. Assim, a identificação de alvos por direção foi separada em subcomportamentos, um para cada direção, onde o grau de prioridade da resposta é dado em função da proximidade do alvo.

O comportamento "EvitarObst"verifica se o agente está próximo de algum obstáculo que possa impedir a sua movimentação. Se for identificado um obstáculo adjacente em qualquer direção, o agente gera uma ação que o move para uma direção livre, evitando colisões.

Essa composição do comportamento "Recolher"torna o agente mais adaptativo e permite que ele tome decisões com base nas informações disponíveis, priorizando a aproximação a alvos, evitando obstáculo e realizando a exploração do ambiente de forma aleatória.

3.1.6 Parte 6

Com o começo da fase 6 do projeto, iniciamos o estudo de procuras em espaços de estados. No entanto, antes de sermos capazes de implementar um mecanismo de procura em espaço de estados, precisamos de ser capazes de exprimir o domínio do problema de modo que a nossa procura seja capaz de o interpretar e utilizar.

3.1.6.1 Domínio do problema

Para definir o domínio do problema, criámos uma biblioteca, **mod**, com 3 classes distintas: Estado, Operador e Problema.

A classe **Estado** é representativa de um estado possível de existir. São identificados por um *idvalor*, que será usado como termo de comparação com outros estados.

O **Operador** representa as operações que se pode aplicar sobre um estado de modo a alcançar outro estado diferente. Também informa do custo da mudança de estado.

O **Problema** representa o problema para o qual se vai tentar encontrar solução. É constituído por um estado inicial e uma lista de operadores que podem ser usados para a chegar a uma resolução.

3.1.6.2 Mecanismo de procura

O mecanismo de procura começa por ser inicializado com uma fronteira vazia. De seguida, recebe um Problema que contem um No com o estado inicial e o conjunto de operadores e começa o processo de exploração.

O **No** é um objeto que encapsula um estado e possui outros dados relevantes à procura, como a profundidade e custo e no antecessor. É desta forma que, caso seja encontrada

uma **Solução**, construida somente por um no final, é possível se retroceder para os nós anteriores até termos o percurso de solução completo.

É de notar que a ordem como os nós são removidos da fronteira depende unicamente da implementação da classe Fronteira. Além disso, o mecanismo tem um método abstrato "memorizar"que permite que implementações deste mecanismo de procura filtrem que nos devem ser inseridos na fronteira. Isto estas características permitem a implementação deste mecanismo com diversos métodos de procura diferente.

3.1.6.3 Procura em profundidade

Para implementar o mecanismo de procura em profundidade, como visto na componente teórica, basta que o nó removido da fronteira seja sempre o mais recente. Para isso, inicializamos uma instância do mecanismo de procura com uma implementação de fronteira em que a remoção segue o princípio *Last In First Out* (LIFO). Uma vez que a procura de profundidade não realiza nenhuma verificação dos nos inseridos na fronteira, o método "memorizar simplesmente adiciona os nos a fronteira.

3.1.6.4 Procura em largura

Começamos por implementar um novo tipo de procura, a "ProcuraGrafo". Esta classe funciona de forma similar ao mecanismo de procura, da qual estende, com a exceção de que, sempre que adiciona um novo nó à fronteira, guarda o estado desse mesmo nó internamente. Essa lista de estados será posteriormente utilizada para impedir que se insiram na fronteira nós com estados que já foram ou já irão ser explorados.

Por fim, a procura em largura é implementada por meio de uma classe que estende de ProcuraGrafo e utiliza uma implementação de fronteira em que a remoção segue o princípio *First In First Out* (FIFO).

3.1.7 Parte 7

Nesta fase além de implementarmos mais mecanismos de procura estudados em aula, adicionámos um módulo de benchmarking ao mecanismo de procura, que nos permite verificar a quantidade de nós guardados e processador durante a procura.

3.1.7.1 Procura em profundidade limitada

Como visto na componente teórica, a procura em profundidade limitada utiliza as bases da procura em profundidade e aplica uma restrição em profundidade. Tendo isso

em mente, a implementação da nossa procura em profundidade limitada vai estender da classe de procura em profundidade, no entanto, terá algumas alterações.

Primeiramente, começamos por aplicar o limite de profundidade. A classe é inicializada com um valor limite para a profundidade, que irá filtrar até que ponto os nos poderão ser expandidos. Por outras palavras, o mecanismo de procura só será capaz de expandir nós cuja profundidade não ultrapasse o valor indicado.

Além de impor o limite na profundidade, esta procura também corrige a suscetibilidade a ficar presa em loops que a procura em profundidade possui. Esse problema é corrigido através da verificação dos nós antecessores. Se ao expandir um no, se verificar que um dos novos nós resultantes possui um estado que já foi observado neste mesmo ramo, é um indicador que estaremos a entrar num loop, e, portanto, esse no não deve ser adicionado a fronteira.

3.1.7.2 Procura em profundidade iterativa

Procura em profundidade iterativa, como indicado, executa a procura em espaços limitados várias vezes, enquanto itera o limite de profundidade.

Na implementação, a classe é inicializada com uma profundidade máxima (*limite_prof*) e profundidade a incrementar (*inc_prof*). Tomando partido da implementação da procura em profundidade iterativa, é feito um loop, onde o index começa em 0, incrementa *inc_prof* a cada ciclo e termina quando chega a *limite_prof*. Neste loop, em cada ciclo é feita a procura em profundidade limitada com o limite equivalente ao index do loop até que se chegue ao limite de profundiade/index ou caso se ache uma solução.

3.1.7.3 Procura melhor primeiro

Para sermos capazes de implementar, primeiro precisamos de ser capazes de ter uma fronteira capaz de ordenar o seu conteúdo em função de uma expressão avaliadora. Para esse fim, criamos a classe FronteiraPrioridade que, recebendo um avaliador no construtor, é capaz de ordenar os nos guardados em memória por via de um *heap*.

Uma vez que a FronteiraPrioriade foi implementada, para criar a ProcuraMelhorPrim bastou criar uma classe, que estende de ProcuraGrafo, que recebe no construtor um avaliador, que posteriormente irá usar para criar a FronteirePrioridade.

Adicionalmente, também alteramos o critério utilizado para decidir se um no deve ser adicionado à fronteira. Se um nó tiver um estado que já foi/vai ser explorado estiver para ser adicionado à fronteira, isso só é possível se o caminho para esse nó for mais eficiente do que o previamente conhecido.

3.1.7.4 Benchmark

Para ser capaz de fazer o benchmarking do método de procura, criei uma classe cujo propósito é guarda dois valores: o número de nós processados e numero máximo de nós em memória durante a execução de uma procura.

Uma instância desta classe é sempre criada dentro do construtor do mecanismo de procura, para poder fazer o benchmarking das procuras feitas por esse mecanismo. O número de nós processados é incrementado sempre que um nó é expandido, enquanto o número de nós e memoria é atualizado sempre que um novo nó é adicionado à fronteira.

Posteriormente, os resultados do benchmarking são escritos na consola sempre que a solução é encontrada ou caso a fronteira fique vazia, demonstrando que não há solução possível.

3.1.8 Parte 8

A fim de terminar o estudo de procura em espaços de estados, implementámos mais dois métodos de procura: procura de custo uniforme e procura informada, juntamente com uma aplicação de teste que permite verificar o funcionamento dos vários métodos de procura criados.

3.1.8.1 Procura custo uniforme

A procura de custo uniforme nada mais é do que uma instância da procura melhor primeiro com uma função avaliadora que avalia os nós em função do seu custo. Resumidamente, este método de procura terá sempre preferência pelos nós com um menor custo.

3.1.8.2 Procura Informada

A fim de implementar a procura informada, criamos uma classe que estende de procura melhor primeiro, mas que altera a procura, recebendo uma heurística, que de seguida fornecerá ao avaliador. Desta forma possibilitamos que, a cada procura, se possa configurar uma heurística diferente para ser utilizada na fronteira. De seguida, a partir desta classe, implementamos um novo tipo de procura informada.

A Procura AA é um tipo de procura informada inicializada com o AvaliadorAA. Este avaliador calcula a prioridade de cada nó baseado no seu custo, juntamente com o valor heurístico.

3.1.8.3 Planeador de trajetos

O planeador de trajetos é uma aplicação que fornece um ambiente de teste para os métodos de procura. É constituído por três conceitos principais:

- **Ligações**: Definidas por um estado inicial, estado final e custo.
- **Problema de planeamento**: Implementa a classe Problema da biblioteca do domínio do problema. Define o problema para o qual se deve tentar arranjar solução. Recebe um estado inicial, estado final e lista de ligações existentes.
- **Planeador de trajetos**: Tem a função de planear um trajeto-solução, tendo em consideração o problema de planeamento e método de pesquisa indiciado.

Quando se deseja realizar um planeamento, o planeador de trajetos recebe uma lista de ligações, a localização inicial e a localização final. Com base nessas informações, é criada uma instância de ProblemaPlanTraj, que é uma implementação da classe Problema do modelo do problema.

Internamente, o ProblemaPlanTraj transforma a lista de ligações numa lista de Operadores (OperadorLigação) e as localizações em Estados (EstadoLocalidade). Uma vez que tem o problema mapeado em objetos do modelo do problema, é capaz de inicializar a classe super, que será a classe Problema do modelo do problema.

Internamente, o objeto ProblemaPlanTraj realiza uma transformação da lista de ligações numa lista de operadores do mod, e as localizações são representadas como estados (EstadoLocalidade), também do mod. Essa conversão permite mapear o problema em objetos do modelo do problema. Com o problema devidamente mapeado, o objeto ProblemaPlanTraj é capaz de inicializar a classe super, Problema, que corresponde à classe base do modelo do problema.

Isto possibilita a utilização do problema, como parâmetro de entrada dos métodos de procura, que irão retornar o percurso solução caso este exista.

3.1.9 Parte 9

Na 9ª fase do projeto, o objetivo principal é iniciar o desenvolvimento de um controlo deliberativo. Com base na teoria estudada, percebemos que o controlo requer uma

representação do modelo do mundo em memória, bem como um mecanismo capaz de interpretar essa representação a fim de realizar as etapas de deliberação e planeamento.

A classe ModeloMundo desempenha um papel fundamental no controlo deliberativo, pois é responsável por armazenar em memória todas as características relevantes do ambiente. A classe mantém informações importantes, como o estado atual do agente, a lista de todos os estados possíveis, a lista de operadores disponíveis e a lista de elementos presentes no ambiente. Além disso, a classe oferece funcionalidades adicionais que podem ser úteis para o controlo, por exemplo, possui um método para calcular a distância entre dois estados no ambiente.

Um aspeto importante da classe ModeloMundo é a capacidade de ser atualizada com base na perceção do agente. Essa atualização ocorre para manter as informações do modelo em concordância com as condições atuais do ambiente. Ao receber uma perceção, a classe atualiza os dados do modelo. Essa atualização é marcada pela alteração da flag "alterado", que indica se houve alguma mudança significativa no modelo em memória após receber a perceção.

Para implementar a funcionalidade de deliberação, desenvolvemos um mecanismo deliberativo que recebe como entrada um ModeloMundo, que utiliza para examinar as informações disponíveis do ambiente e identifica os objetivos do agente.Uma vez identificados os objetivos, o mecanismo deliberativo cria uma lista que contem esses objetivos e realiza uma ordenação com base na distância. A ordenação por distância permite determinar quais objetivos estão mais próximos do agente, o que pode influenciar a prioridade ou a sequência de ações a serem realizadas.

Por fim, começamos a desenvolver o controlo deliberativo. O controlo deliberativo segue o processo de tomada de decisão estudado:

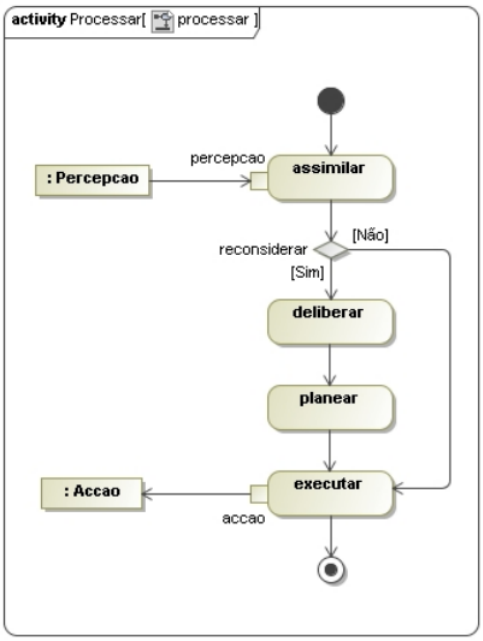

Figura 3.2: Controlo deliberativo

O primeiro passo é atualizar o modelo mundo através de uma perceção. De seguida, verifica a flag de alteração no modelo mundo. Caso, após a atualização, tenham ocorrido alterações no modelo mundo, terá que ocorrer uma redeliberação.

Caso ocorra uma redeliberação, o controlo utiliza o mecanismo de deliberação, mencionado anteriormente, para obter uma lista atualizada com os objetivos ordenamos por proximidade. Uma vez que tem a lista de objetivos, utiliza o planeador para traçar um percurso para chegar ao alvo mais perto. É de notar que a este ponto, ainda não temos o planeador implementado, sendo esse o foco da fase seguinte. Assim que se tem o trajeto definido, o controlo executa o trajeto, obtendo assim a ação a executar.

Se não ocorrer redeliberação, o controlo passa logo para a etapa de execução, utilizando os objetivos e plano traçados na última iteração.

Este ciclo irá manter-se até que todos os objetivos tenham sido alcançados, ponto esse em que o agente ficará imóvel, incapaz de traçar qualquer plano de ação.

3.1.10 Parte 10

Para terminar o controlo deliberativo, começado na última fase, assim como o estudo de agentes deliberativos, esta fase tem o objetivo de implementar o mecanismo de planeamento do controlo deliberativo.

Antes de criar o planeador, é necessário definir a estrutura que representará o plano traçado. Para isso, foi desenvolvida a classe PlanoPEE. Esta classe é inicializada com uma solução, que vai representar o percurso que o agente deve seguir para alcançar o objetivo. Para fornecer o trajeto ao agente, a classe PlanoPEE possui um método que recebe um estado como entrada. Esse método verifica se o estado fornecido faz parte da solução do percurso solução. Caso o estado faça parte da solução, o método retorna o operador que levará o agente para o próximo estado na do percurso.

Desta forma, processo do PlaneadorPEE torna-se bastante simples. Primeiro, verifica se há objetivos na lista. Caso existam, ele retira o primeiro objetivo da lista, que representa o objetivo mais próximo ao agente. Em seguida, utiliza um método de procura para encontrar uma solução que leve o agente até esse objetivo. Uma vez obtida a solução, o PlaneadorPEE cria uma instância da classe PlanoPEE, utilizando a solução encontrada como parâmetro de inicialização. Essa instância de PlanoPEE é então retornada pelo PlaneadorPEE.

Para método de procura, utilizámos a procura informada AA com uma heurística de distância de Manhattan, uma vez que não são possíveis movimentos diagonais.

3.1.11 Parte 11

Esta fase foi dedicada a revisão do código implementado anteriormente, pelo que não há nada a relatar.

3.1.12 Parte 12

A fase final do nosso projeto foi a implementação de um planeador que utiliza processo de decisão de Markov. Na implementação do planeador baseado no processo de decisão de Markov, é necessário desenvolver mecanismos que permitam calcular a utilidade de cada estado e, a partir disso, determinar a política de ações a ser seguida pelo agente.

3.1.12.1 Calculo da utilidade

A classe MecUtil serve como o mecanismo responsável por calcular e mapear os graus de utilidade de cada estado no modelo mundo. A classe é inicializada com o modelo mundo, o valor de desconto *gamma* e o valor máximo *delta_max*.

No processo utilizado para calcular a utilidade de cada estado, através o processo de decisão de Markov, a utilidade de um estado é determinada com base nas utilidades dos estados futuros.

Para o cálculo de utilidade é utilizado um loop que percorre todos os estados presentes no modelo mundo. Para cada estado, são aplicados todos os operadores e, posteriormente, definimos a utilidade dos estados futuros, de um dado estado, a partir da utilidade do estado seguinte com a maior utilidade. Por fim, aplicamos o desconto *gamma* a utilidade dos estados futuros e somamos a recompensa imediata, caso exista, para obter o grau de utilidade de um estado.

É importante mencionar que dependendo do tamanho do ambiente, pode ser preciso adaptar o valor de *gamma*. Caso o ambiente tenha uma dimensão muito grande, e o desconto tenha um valor elevado, a poderá chegar a um ponto que o desconto tenha sido aplicado tantas vezes que, devido ao seu elevado valor, suprima quase por completo o valor da utilidade.

Este loop é executado várias vezes para cada estado de modo a aumentar a precisão do cálculo da utilidade. O número de vezes que este loop é executado está relacionado ao limite de uma "convergência", definido por *delta*. Resumidamente, quanto maior o valor máximo de *delta*, menos loops serão executados. Embora isso alivie a carga temporal e computacional do cálculo, também resulta em valores de utilidade menos precisos.

Como resultado, o mecanismo devolve um dicionário que associa cada estado a um grau de utilidade.

3.1.12.2 Definição da política

A definição da política é feita por via da classe PDM. Recebe como parâmetros de entrada um modelo mundo, *gamma* e *delta_max*.

A política determina a ação a ser tomada em cada estado para obter o melhor estado sucessor. Para isso, é necessário ter mapeados os graus de utilidade dos estados. Para realizar essa tarefa, criamos uma instância do mecanismo de utilidade e geramos um dicionário estado-utilidade.

Com o mapa de utilidades disponível, percorremos todos os estados e procuramos o operador que leva ao estado sucessor com o maior grau de utilidade. Dessa forma, a política resultante é um dicionário que associa cada estado a um operador e ao seu respetivo grau de utilidade.

3.1.12.3 Planeador PDM

Assim como ocorreu quando implementámos o planeador da fase passada, antes de fazer o PlaneadorPDM precisamos de antes definir o plano que este vai retornar.

A classe PlanoPDM é uma implementação de Plano e recebe o mapeamento de utilidades e políticas. Quando uma ação é solicitada ao PlanoPDM, o estado fornecido é utilizado como chave no dicionário de política, permitindo assim obter a ação que leva ao melhor estado sucessor. Desta forma, o PlanoPDM utiliza a política mapeada para determinar a ação a ser tomada com base no estado atual. O mapeamento de utilidades recebido é apenas utilizado para a componente visual do programa.

Concluímos então que a tarefa do Planeador PDM é simples. Caso existam objetivos, ele inicializa uma instância da classe PDM, que recebe o modelo do mundo e os objetivos como entrada. A partir dessas informações, o PDM gera o mapeamento de utilidades e políticas, utilizados para criar o PlanoPDM.

4

Revisão do projeto realizado

4.1 Introdução

Neste capítulo, serão abordados os principais desafios enfrentados ao longo do projeto, incluindo questões teóricas e práticas que surgiram, as correções realizadas e os detalhes relevantes. Dado que nem todas as fases do projeto apresentaram problemas significativos na resolução/submissão, apenas algumas delas serão mencionadas neste capítulo, que será focado nas mais relevantes.

É de notar que podem ocorrer discrepâncias entre os problemas relatados e as fases a eles associados. Isto acontece porque, uma vez que a pagina de submissão das fases possui o acesso vedado, a única referência disponível para consultar os erros e as devidas correções é o histórico de *GitHub* do repositório utilizado para o projeto, onde pode acontecer que alguns erros ou fases não estejam devidamente identificados.

4.1.1 Parte 1 - Entrega incompleta

Na primeira fase do projeto, não fui capaz de completar o método percecionar da classe Personagem. O método foi implementado posteriormente e enviado, juntamente com a segunda fase.

4.1.2 Parte 5 - Entrega não entregue e com lapsos

A aula de submissão da quinta parte do projeto foi a primeira e ultima aula da cadeira de IASA à qual não fui capaz de comparecer. Isto significa que todo o trabalho desta fase teve que ser realizado em casa, sem qualquer tipo de suporte.

Colocando o problema com a submissão de parte, inicialmente também tive alguma dificuldade em corrigir um erro que permitia que o agente colidi-se com o obstáculo uma primeira vez antes de ser capaz de se desviar. Isto aconteceu porque interpretava de maneira errada a informação contida na perceção. Após ler bem a documentação da biblioteca *sae* e olhar para as classes fui capaz de arranjar o erro sem grandes problemas.

Foi também nesta fase que recebi um email do professor regente da cadeira a relatar que o meu projeto não se encontrava conforme o pedido. Isto aconteceu, pois, uma vez que não fui capaz de comparecer à aula, não soube que haviam certas pastas que deveriam ter sido criadas e que não estavam contidas no enunciado.

4.1.3 Parte 7 - Sintaxe dos nomes corrigidos

Foi nesta fase que me apercebi que não nomeava as classes, métodos e objetos conforme a sintaxe PEP 8, utilizada em Python. Isto deve-se ao facto de nunca ter utilizado a linguagem Python anteriormente e, portanto, utilizava a sintaxe de Java. Todos os nomes foram alterados para respeitar o PEP 8.

Para além do problema com a sintaxe dos nomes, também tive bastantes erros, muitos deles derivados de fases passadas, que não me possibilitaram entregar o projeto a funcionar. Alguns dos erros foram os seguintes:

- Invocação métodos internos do Python de maneira errada.
- Incoerências no número de '_' fazia com que certas funções não fossem identificadas pelo compilador.
- Engano em perceber que estruturas de dados eram mais indicadas em cada módulo do projeto.
- Erros de lógica.
- Comparações feitas sem utilizar __eq__.

Após uma revisão completa do projeto e dos enunciados fui capaz de identificar os erros e corrigi-los de modo a entregar a fase seguinte a funcionar.

4.1.4 Parte 10

Foi no decorrer da fase 10 que foi percebido que o avaliadorAA devidamente implementado, componente que era essencial para o planeamento automático. Foi implementado imediatamente sem grandes problemas.

Foi também adicionada posteriormente a verificação da existência de objetivos em vários componentes do planeador.

4.1.5 Parte 12

A implementação do processo de decisão de Markov foi onde foram encontradas mais dificuldades de implementação. A fase entregue estava muito incompleta, muito devido à dificuldade de compreensão da teoria por detrás do mecanismo. Mais tarde, após rever a componente teórica e compreender os princípios do processo de decisão, consegui implementar o mecanismo.

Embora tenha consigo implementar o mecanismo, deparei-me com uma solução extremamente ineficiente, demorando bastante tempo, na grandeza de minutos, a realizar cada movimentação. Isto deveu-se ao facto de estar a computar a aplicação dos operadores a todos os estados do modelo em várias etapas do processo de planeamento.

A solução foi, no modeloPDMPlan, aplicar logo todos os operadores aos estados do modelo e guardar os resultados num dicionário. Assim, sempre que fosse necessário consultar o resultado da aplicação de um operador sobre um estado, basta consultar o dicionário. Deste modo essa computação é feita uma única vez em todo o planeamento.

5

Conclusão

Ao concluir este projeto, pude adquirir um conhecimento mais abrangente da área de inteligência artificial e as suas aplicações. Através da implementação de diferentes abordagens e algoritmos, obtive um entendimento mais sólido dos conceitos teóricos e das técnicas utilizadas na construção de agentes inteligentes.

Além disso, também gostei bastante da forma como o projeto estava organizado. As entregas contínuas e semanais davam a oportunidade de aplicar e interiorizar os conhecimentos lecionados em sala de aula. Além disso, a avaliação continua forçou o interesse a acompanhar as aulas e manter um ritmo e trabalho e engajamento continuo.

No geral, este projeto permitiu-me aprofundar o meu conhecimento em inteligência artificial e desenvolver habilidades na implementação de agentes inteligentes.

6

Referencias

As imagens foram obtidas dos slides teóricos da cadeira de IASA, lecionada no ISEL.

- https://2223moodle.isel.pt/pluginfile.php/1197058/mod_resource/content/2/07-arq-react-1.pdf
- https://2223moodle.isel.pt/pluginfile.php/1197236/mod_resource/content/3/11-pee-1.pdf
- https://2223moodle.isel.pt/pluginfile.php/1204525/mod_resource/content/2/14-arq-delib.pdf
- https://2223moodle.isel.pt/pluginfile.php/1197059/mod_resource/content/1/P03-iasa-react-1.pdf
- https://2223moodle.isel.pt/pluginfile.php/1204526/mod_resource/content/3/P09-iasa-arq-delib.pdf