INTELIGÊNCIA ARTIFICIAL PARA SISTEMAS AUTÓNOMOS

Relatório Final

Pedro Azevedo 47094
Licenciatura Engenharia Informática e Multimédia
Engenheiro Luís Morgado
6 julho 2022

Índice

1. Introdução

O projeto realizado para a Unidade Curricular Inteligência Artificial para Sistemas Autónomos, teve por base os temas relacionados com Inteligência artificial e seguindo os método e princípios da de desenvolvimento de engenharia de software.

No projeto, desenvolveu-se Arquiteturas para diversos tipos de Agentes Inteligentes tais como, Agentes Reativos e Agentes Deliberativos. Para se conseguir implementar este tipo de agentes, estudou-se e implementou-se métodos e mecanismos que fazem com que estes agentes consigam funcionar bem. Desenvolveu-se uma biblioteca com estes mecanismos, como máquinas de estados, componentes de um problema, mecanismos de procura e mecanismos de aprendizagem.

Nas primeiras duas partes do projeto, implementou-se na linguagem Java, e o restante projeto foi implementado com a linguagem Python. Todo o projeto foi desenvolvido utilizando o software *Visual Studio Code*. Uma biblioteca que foi usada na maior parte do projeto foi a Simulador de ambientes de execução, *sae*, onde permitiu fazer uma simulação, utilizando os módulos e bibliotecas desenvolvidas, dos agentes de uma forma visual.

Este relatório apresenta inicialmente um enquadramento teórico que foi a base do projeto, teoria estudada ao longo do semestre, onde, para a realização, teve se por base principalmente os documentos disponibilizados pelo Engenheiro Luís Morgado, disponíveis na plataforma *Moodle*. De seguida apresenta se a descrição da concretização dos temas estudados e falados no enquadramento teórico, que está dividido em partes/secções relacionadas com os temas estudados e apresentados ao longo do semestre. Como o projeto foi realizado durante o semestre, possíveis erros de falta de conhecimento podem ter sido cometidos, tais erros são apresentados no capítulo 4. Por fim, o relatório termina com uma conclusão do trabalho e da Unidade Curricular, seguido da *Webgrafia*.

2. Enquadramento Teórico

2.1. O que é Inteligência Artificial?

Segundo o artigo de John McCarthy (What is Artificial Intelligence? 24 novembro 2004, página 2, documento 1.1 da *Webgrafia*), Inteligência Artificial é a ciência e engenharia de fazer tanto máquinas inteligentes como programas informáticos inteligentes. Relaciona-se com a tarefa semelhante de compreender a inteligência humana, contudo não se limita a métodos que são biologicamente observáveis.

Stuart Russell e Peter Norvig definiram e diferenciaram quatro objetivos e definições potenciais de Inteligência Artificial, que distinguem os sistemas informáticos com base no raciocínio e pensamento com a ação. Por um lado, existe a abordagem humana, onde queremos que os sistemas pensem e atuam como humanos. Por outro lado, a abordagem ideal, onde se pretende que os sistemas pensem e atuem de forma racional. (com base documento 2 da *Webgrafia)*

Podemos afirmar que, a inteligência artificial é a junção entre duas matérias que hoje me dia são muito exploradas, a informática e dados, para assim, obtermos a resolução de problemas. Para esta resolução de problemas, podemos ver duas perspetivas, analítica e sintética. São necessárias as ciências empíricas como a biologia, a neurobiologia, e ciências que se focam na mente humana, para a parte mais analítica, e é necessário a ajuda das ciências cognitivas, e os seus princípios e teorias, formando também modelos sintéticos.

Existem três paradigmas de inteligência artificial, o simbólico, o conexionista e o comportamental.

O paradigma simbólico, onde um sistema de símbolos físicos tem os meios necessários e suficientes para a atividade inteligente em geral (Luís Morgado, Inteligência Artificial Introdução, documento 3 da *Webgrafia*). Ou seja, para a produção de informação, de dados, de respostas, é necessária uma estrutura de símbolos que representem a realidade. É construída uma representação de conhecimento através de relacionamento, uma rede semântica. É com o conhecimento que obtemos sistemas inteligentes, claro, com base em dados e informação.

O paradigma *conexionista*, é a abordagem que procura compreender o cérebro humano ao nível neural, em que o objetivo é o de perceber como aprendemos e nos lembramos, e por isso, este paradigma é referido como computação neuronal, semelhante aos neurónios. Em 1943 Warren McCulloch e Walter Pitts criaram um modelo computacional para redes neuronais baseadas em matemática e algoritmos como a *threshold logic*. (documento 5 da *Webgrafia*)

O paradigma comportamental diz que os seres humanos são compreendidos como organismos comportamentais, seres complexos na natureza e na capacidade mental, porém, são melhor compreendidos através de uma análise da resposta aos estímulos. Podemos ver como exemplo, a atribuição de recompensas e castigos dependendo do seu comportamento, através dos quais, o ser humano aprende o que deve e não deve fazer. Este paradigma baseia-se também no comportamento emergente dos agentes, da interação dos indivíduos uns com os outros e a sua adaptação. (documento 6 da *Webgrafia*)

2.2. Sistemas Autónomos

Para se considerar algo autónomo, esse algo tem que ser independente, funcionar sozinho, sem ajuda, é algo autossuficiente, que produz tudo o que precisa para se sustentar, para viver, para funcionar.

Por isso, um sistema autónomo é um sistema que funciona sozinho, sem ação de terceiros. Por exemplo, um carro autónomo. Um carro autónomo, ou seja, um carro que não precisa de um condutor para o guiar, é um sistema autónomo por essa mesma razão. O carro consegue aperceber-se do mundo e reagir dependendo da situação, sempre com uma finalidade.

Um sistema autónomo, como o nome indica, tem autonomia. A autonomia é a capacidade de atuar por iniciativa própria e independente de terceiros. Porém, ser autónomo não significa que seja inteligente, pois para ser inteligente tem que se saber escolher a melhor opção na altura de solucionar um problema. Um sistema inteligente tem como característica a autonomia, pois reage as situações por iniciativa, e como é inteligente resolve o problema da melhor maneira possível.

2.3. Agente Inteligente

Um agente é a representação computacional de um sistema autónomo inteligente, como por exemplo, um trator agrícola que lavra a terra num ambiente real, ou por exemplo uma personagem (NPC – *Non-Player Character*) que tem a tarefa de perseguir o jogador num ambiente virtual. Podemos concluir que estes dois exemplos têm algo em comum, executam uma tarefa num ambiente. Os humanos têm olhos, ouvidos, têm sensações que fazem percecionar num certo ambiente, e com a informação que captamos podemos pensar numa solução, ou numa ação para executar, e assim o fazemos com o nosso corpo. E tal como os humanos, os agentes inteligentes também têm estes *inputs*, como sensores, câmaras, e através da informação captada, o agente pode processar e, dependendo do ambiente (real ou virtual), pode atuar, através de rodas, vídeo, altifalantes, escrita, executar um programa, executar uma ação no *software* ou *hardware*, movimentar-se com as suas pernas virtuais pelo mundo virtual.

Podemos dizer que um agente inteligente é autónomo, reativo, pró-ativo e até sociável. Contundo, para ser inteligente, tem que saber raciocinar, tem que aprender, conhecer, saber analisar uma situação e retirar a melhor opção e a opção correta com base no que sabe, no que aprendeu e até mesmo com base no que já falhou.

Um agente que é inteligente tem que ser, portanto, um agente racional, escolhe e atua com a ação correta, a ação que maximiza o objetivo, o valor, através do conhecimento que já adquiriu.

Existe como opção o agente ter um modelo reativo ou deliberativo. Por um lado, um agente com uma arquitetura reativa, tem a perceção do mundo atual e após processar, reage com uma ação, tem objetivos implícitos. Por outro lado, o agente pode seguir uma arquitetura deliberativa, ou seja, em vez de percecionar e atuar, este agente delibera antes de agir, volta a percecionar e até alterar a ação que ia executar pois houve uma alteração no ambiente. Podemos retirar como um exemplo um carro autónomo, o sinal está vermelho, e quando este altera para verde, o carro vai acelerar. O sinal já está verde, contudo aparece um peão a atravessar a estrada atrás de uma bola, o carro tem que conseguir voltar a percecionar o mundo para ver que agora não pode avançar.

2.4. Modelos de Dinâmica

Um modelo de um sistema computacional está organizado em 3 partes, a estrutura, a dinâmica e o comportamento. A estrutura, denota as partes e as relações entre partes de um sistema, é a organização no espaço, a memória. Vendo a *Figura 1* (retirada do documento *Modelos de Dinâmica*, Luís Morgado, página 2) , é o Estado, a configuração do sistema relevante para caracterizar a sua dinâmica.

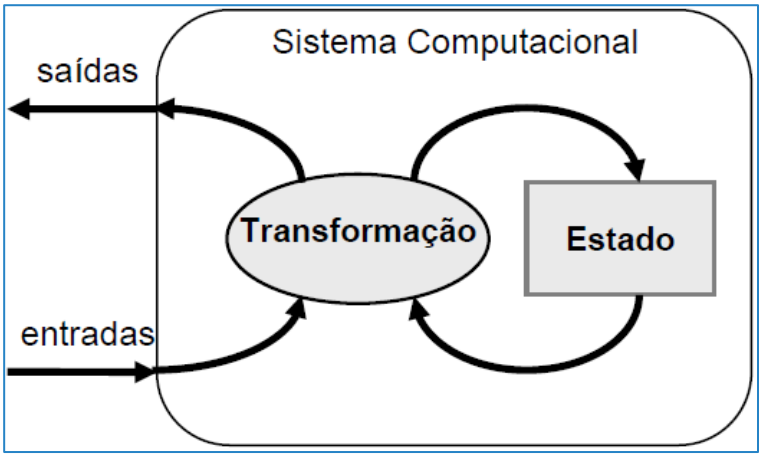

Figura 1 - Organização de um sistema

A dinâmica é a organização no tempo que denota a forma como as partes e as relações entre partes de um sistema evoluem no tempo. Equivalente a parte da Transformação na figura acima.

Por fim, o comportamento detona a forma como o sistema age ou reage, como interage dependendo dos estímulos e perceções que recebe do ambiente onde reside. Ou seja, expressa a estrutura e a dinâmica do sistema.

A dinâmica vai ser uma função onde recebe como variáveis as entradas e o estado atual e produz um estado seguinte e saídas. Esta função, é a função Transformação. A função transformação é composta por duas funções diferentes, sendo que uma resultará o estado seguinte e a outra a saída, Podendo chamar como Função de Transição de Estado, e a Função de Saída, respetivamente.

Esta organização, este modelo que representa um mecanismo computacional, é designado por Máquina de Estados, sendo que contém um número de estados finito. Aqui, existem duas maneiras de calcular a função de saída, a Máquina de *Mealy*, onde a função de saída depende das entradas, e a Máquina de *Moore*, que em oposição da outra, é independente das entradas. Ou seja, por um lado, o ambiente onde o agente se encontra afeta a saída, que podem ser variáveis que influenciam depois na decisão de certos eventos. Por outro lado, na Máquina de *Moore*, o ambiente não afeta para gerar a função de saída.

2.5. Engenharia de *Software*

Para se desenvolver *software*, não basta saber programar, não basta saber as linguagens de programação, é necessário aplicar abordagens sistemáticas, disciplinadas e quantificáveis ao desenvolvimento, operação e manutenção de *software*.

Hoje em dia, desenvolvem-se aplicações cada vez mais complexas, com cada vez mais informação, o que cria um problema da complexidade. O problema é a complexidade estrutural, existe um problema de interação, do sistema para o sistema, de elementos de informação e até entre os elementos das equipas de desenvolvimento. Existe uma explosão combinatória, existe um crescimento exponencial da complexidade.

Podemos ter dois tipos de complexidade, a Desorganizada e a Organizada. Por um lado, a desorganizada resulta do número e heterogeneidade das partes de um sistema, estes podem ter interações irregulares entre si, e as características globais do sistema podem ser inferidas com base em métodos estatísticos. Por outro lado, a complexidade é organizada tendo um fim em vista, resulta de padrões de inter-relacionamento entre as partes, com as interações que obedecem a padrões correlacionáveis no espaço e no tempo.

Então, a Engenharia de *Software* vem para reduzir e controlar a complexidade, seguindo certas métricas, princípios e padrões. Métricas como o Acoplamento, Coesão, Simplicidade e Adaptabilidade.

O Acoplamento está relacionado com o grau de interdependência entre subsistema, tendo um carácter inter-modular. Estas relações podem ser mais fortes do que outras, objetos que herdam ou compõem outros objetos, tem uma relação mais forte do que objetos que estão associados ou que tem uma relação de dependência. A Coesão, o nível de coerência funcional de um subsistema, tendo uma característica intra-modular. A Simplicidade, corresponde ao nível de facilidade de compreensão da arquitetura, para que outros a possam entender, e por fim a Adaptabilidade, o nível de facilidade de fazer alterações da arquitetura, incorporando novos requisitos.

Os princípios que se deve ter quando se está a construir uma arquitetura são a Modularidade, a Factorização e a Abstração. O princípio da modularidade consiste na decomposição de um sistema em partes correlacionadas para assim se sistematizar interações e lidar com a complexidade combinatória. Consiste também na eliminação de redundância e garantia de consistência, não repetir o que já foi feito, com factorização. Ainda na Modularidade, pode-se falar também do encapsulamento, ou seja, isolar detalhes internos de um sistema das relações com o exterior, com a finalidade de reduzir interligações, dependências, relacionar estrutura e função no contexto de uma parte, obtendo acesso exclusivo através das interfaces disponibilizadas, interfaces estas que são como contratos funcionais para interação com o exterior.

Uma das principais causas para falhas e *bugs* durante o desenvolvimento de *software*, é a redundância, e o princípio da factorização vem reduzi-la. O objetivo é quando se tem partes, por exemplo de código, iguais em vários locais da arquitetura, colocar essas partes de forma que só esteja num local e seja acedido por todas as partes que precisem. Dois mecanismos para executar este princípio é utilizando Herança, onde tem um nível de acoplamento alto, e que um objeto herda os métodos de outro, fazendo com que os métodos não se repitam nos objetos. O outro mecanismo é a delegação, tendo um acoplamento baixo, e onde um objeto utiliza outro, é delegado um método ao outro objeto. Por fim, o último princípio é a abstração, sendo esta uma ferramenta bastante importante para baixar a complexidade, pois identifica-se características comuns a diferentes partes, realçando o que é essencial. Com isto, é criado objetos com métodos abstratos que têm partes e características comuns a vários objetos, e estes utilizam esses métodos, podendo adicionar partes que sejam diferentes dos outros.

A representação abstrata de um sistema é chamada de Modelo, sendo um meio para lidar com a complexidade, obtendo e sistematizando progressivamente conhecimento. Um modelo foca em aspetos que são importantes no sistema omitindo o resto, tem facilidade na transmissão e compreensão das ideias envolvidas, é claro e objetivo para que se perceba bem, sendo também prático e eficaz tendo uma representação correta e rigorosa, sendo possível retirar conclusões corretas do sistema descrito.

2.6. Arquitetura de Agentes Reativos

Um dos modelos estudados foi o Modelo Reativo. Com este modelo, um agente tem uma perceção e é desencadeada uma ação, contém objetivos implícitos. Como vemos na *Figura 2* (retirada do documento Arquitectura de Agentes Reactivos Parte 1, Luís Morgado, página 6), o agente recebe informação do ambiente, uma perceção, tem uma reação relacionada com o que percecionou e atua no ambiente. Como por exemplo, um robô que aspira, percorre um espaço de uma certa forma, mas quando do ambiente perceciona algo, uma parede por exemplo, este reage com uma ação de virar para o lado contrário.

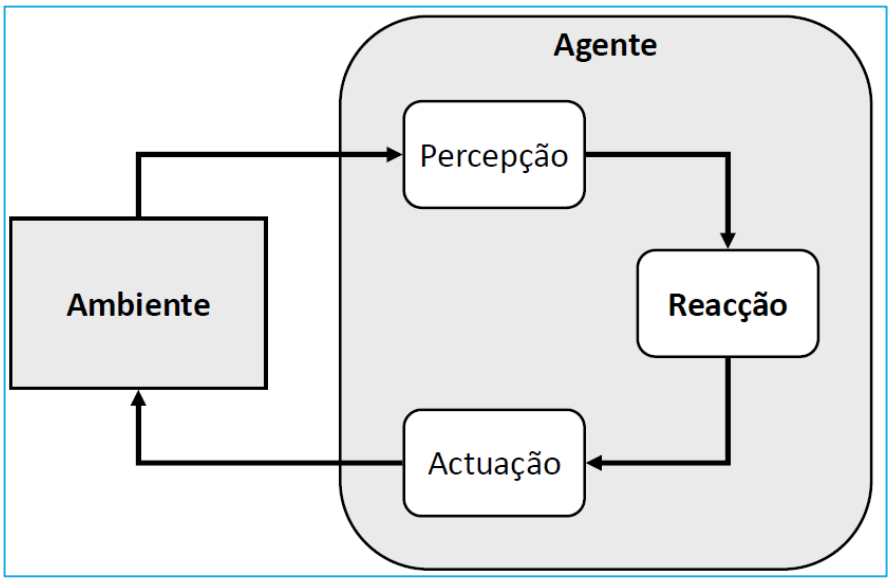

Figura 2 - Arquitetura reativa

Após o agente percecionar, este vai ter uma reação. Neste mecanismo de reação, as ações são aplicadas de forma instantaneamente, são reações fixas e predefinidas para cada estímulo do ambiente, para cada evento do ambiente existe uma ação definida previamente, e no instante que o agente perceciona esse evento o agente aplica logo a ação relaciona. Portanto, perceção causa estímulo tendo uma resposta associada, provocando uma ação. Neste caso, o agente não guarda uma "imagem" do ambiente, ou seja, não guarda na memória uma representação do mundo.

No tópico anterior referiu-se que um princípio para uma boa arquitetura, para um bom desenvolvimento de *software* para diminuir a complexidade, é a modularização. Por isso, para os modelos reativos, podemos modularizar os conjuntos de reações, agrupar num módulo comportamental, Comportamento. Este comportamento vai relacionar as perceções com as ações, podendo este ser composto por outros comportamentos.

Sabendo que uma perceção pode despertar várias reções em simultâneo, como escolher a ação a executar?

Respondendo à pergunta acima referida, pode-se ter mecanismos de combinação e seleção de ações, como a execução paralela de ações, onde as ações a executar não interferem umas com as outras, a combinação de ações, que com diferentes perceções percecionadas, combinam-se as ações numa só, e por fim, a precedência de ações, em ações que interferem umas com as outras, vai ser atuada a ação com mais prioridade.

Em relação a coordenação de comportamentos, num comportamento composto, para a seleção da ação, os comportamentos estão organizados numa hierarquia invariável de supressão, sendo que uma reação pode passar por cima de outra e substituir se estiver num patamar mais alto da hierarquia. Como na precedência de ações, as respostas são selecionadas de acordo com uma prioridade associada podendo esta mudar durante a execução. As ações ainda podem ser combinadas numa só, uma fusão de ações, por exemplo com uma soma vetorial para um movimento.

Um agente reativo pode conter um controlo reativo. Neste controlo está definido os objetivos, qual a finalidade do agente, e tem os comportamentos, que têm as formas de concretizar os objetivos, e sub-objetivos. Como por exemplo, no projeto realizou-se um agente cujo objetivo é recolher os alvos, contudo, tem que explorar o ambiente, evitar os obstáculos assim como aproximar-se dos alvos para puder recolher. Aqui tem comportamentos onde se pode mover para Norte, Sul, Este ou Oeste, e consegue percecionar o sistema nessas direções também.

Então, mas quando o agente perceciona, que ação executar? Pois bem, como falado anteriormente, existe uma hierarquia sendo que o agente vai sempre aproximar se do alvo, mas se tiver um obstáculo a frente desvia-se dele, se não tem obstáculos para desviar ou alvos para se aproximar, explora.

Se o agente funcionasse só assim, sem memória, acabaria por explorar localizações já exploradas e ir ter aos locais onde existe um objeto e voltar a evita lo. Estes agentes têm que evitar o passado, caso contrário, podem ficar encravados no mesmo local, ou fazer movimentos e comportamentos cíclico.

Para evitar o mencionado, acrescenta-se memória no controlo, ligando-a com a reação. Assim, o agente pode conter uma representação interna de perceções anteriores, e assim evitar situações conhecidas, com a possibilidade de gerar forças virtuais que impedem de voltar a locais anteriormente visitados. Estando a memória ligada com as reações, é possível manipular o estado, evitar que o agente tenha uma certa reação. Então, o comportamento pode ter um estado, uma memória, criando assim comportamentos com Máquina de Estados, podendo haver assim uma manutenção de estado. Algumas vantagens que vêm com este modelo são, a possibilidade de produzir todo o tipo de comportamento, representar dinâmicas temporais como a evolução do estado ao longo do tempo e obter reações com base no passado, ter a possibilidade também de ter comportamentos mais complexos com a evolução de estado, podendo agir devido a ausência de mudança, e a capacidade de lidar com situações de falha por ausência de exploração de ações no passado. Contudo, é necessário espaço de memória para guardar a memória (de acontecimentos passados) do agente, aumentando, assim a complexidade espacial assim como manter as representações de cada estado, aumentando a complexidade computacional, aumentando a complexidade da arquitetura. Adicionalmente, esta arquitetura reativa não suporta representações complexas nem explora planos alternativos de ações, mantém sempre o predefinido.

É possível que um agente tenha comportamentos organizados por camadas concretizando objetivos de uma forma independente, podendo um comportamento afetar outro comportamento, assumir o controlo de outro comportamento, caso o que controle esteja numa camada superior e o controlado num inferior, sendo que estes últimos não têm conhecimento das camadas superiores. Chamando assim de Arquitetura de Subsunção. Nesta arquitetura, as camadas superiores podem controlar as inferiores impedindo que estas não comuniquem entre módulos, desativando o comportamento, e repor o estado inicial de um comportamento, um *reset* portando.

2.7. Raciocínio Automático

Um sistema computacional consegue resolver um problema, apresentar uma solução com alternativas possíveis de forma autónoma através do que conhece, do que tem na memoria, da experiência, com a capacidade do Raciocínio Automático. Ou seja, um agente recebe informações acerca do ambiente em que se encontra, e através desses *inputs*, consegue resolver problemas, retirar soluções fundamentadas no conhecimento. Este processo, de retirar conclusões a partir do conhecimento, a manipulação da representação de conhecimento a fim de chegar a conclusões, soluções, é denominado de inferência.

Quando um agente está à procura de uma solução, a explorar as diferentes possibilidades, este tem que ter um raciocínio prospetivo, ou seja, que explora as possíveis consequências no futuro, tem que fazer uma cópia interna do ambiente, a chamada representação interna, para ter noção de todas as possibilidades e conseguir a partir daí ter resultados mais precisos, quanto mais informação, mais fácil é de prever o futuro. Existem várias possibilidades de explorar e avaliar as opções, através do custo e ou da utilidade.

A melhor ferramenta para o Raciocínio Automático é ter o Modelo do Problema, ter uma representação. Consegue-se através da utilização dos Estados, que pode tanto representar uma opção para a solução do problema como também um passo para lá chegar, ou seja, é uma situação , configuração na resolução, sendo que cada Estado é único e tem uma identificação única. Para transitar entre Estados para exploração, é necessário o uso de Operados que representam uma ação, pelo que, o conjunto dos Estados e das transições entre eles denomina-se por Espaço de Estados, que é representado sob forma de grafo como vemos na *Figura 3* (retirada de shorturl.at/beoQX)

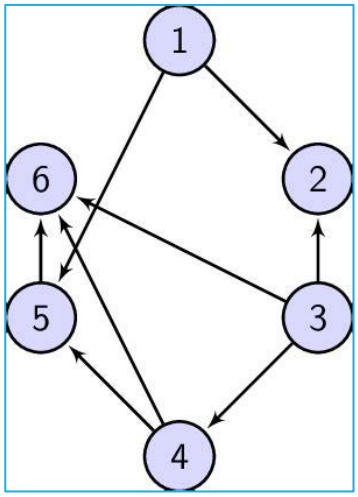

Figura 3 - Grafo - Espaço de Estados

O mais importante para o Raciocínio Automático é ter um problema, este ser constituído por um Estado Inicial, Operadores para sair deste Estado e ir avançando no Espaço de Estados, e o principal, ter um objetivo, estando este em forma de Estado.

Mas como é que é que se vai aplicar os Operadores, um Estado pode transitar para dois estados diferentes para qual transitar? É necessário então haver um Mecanismo de Raciocínio, uma forma de explorar as opções possíveis para encontrar uma solução através de simulação prospetiva, olhado e explorando o futuro, tendo por base uma representação interna do problema.

2.8. Procura em Espaço de Estados

Para se chegar a uma solução de um problema, para ir de um Estado Inicial até ao Estado Objetivo, é necessário procurar no Espaço de Estados, aplicar Operadores para transitar de Estado para Estado. A questão é qual o Raciocínio através de procura, qual o planeamento.

Como se pode imaginar, existem vários processos de procura, processo este que consiste na exploração dos espaços de estados, sendo que cada etapa na procura cada passo que se dá, passa-se pelos chamados Nós (um Nó corresponde a um Estado). Nestes mecanismos constroem-se a Árvore de Procura, onde basicamente começa-se pelo Nó raiz (Nó correspondente ao Estado Inicial) e vai-se colocando os Nós que o sucedem, e depois os Nós que sucedem esses Nós e assim sucessivamente, coloca-se por baixo em forma de árvore os Nós onde se aplica um operador e vai-se para esse Nó na transição. Por fim existe a Fronteira de Exploração, onde tem de forma ordeira os Nós a serem explorados, sendo que a ordem de exploração de Nós, a ordem desta Fronteira, é o que determina a estratégia de controlo de procura.

Podemos observar na *Figura 4* (retirada do documento Procura em Espaços de Estado, Luís Morgado, página 4), do lado esquerdo o Grafo do Espaço de Estados e do lado direito a sua Árvore de Procura. As setas apontam para onde o Estado pode transitar, podendo chamar cada seta de Operador, e na árvore, os Nós a azul são os que já foram explorados, e os restantes estão na Fronteira de Exploração á espera de serem explorados.

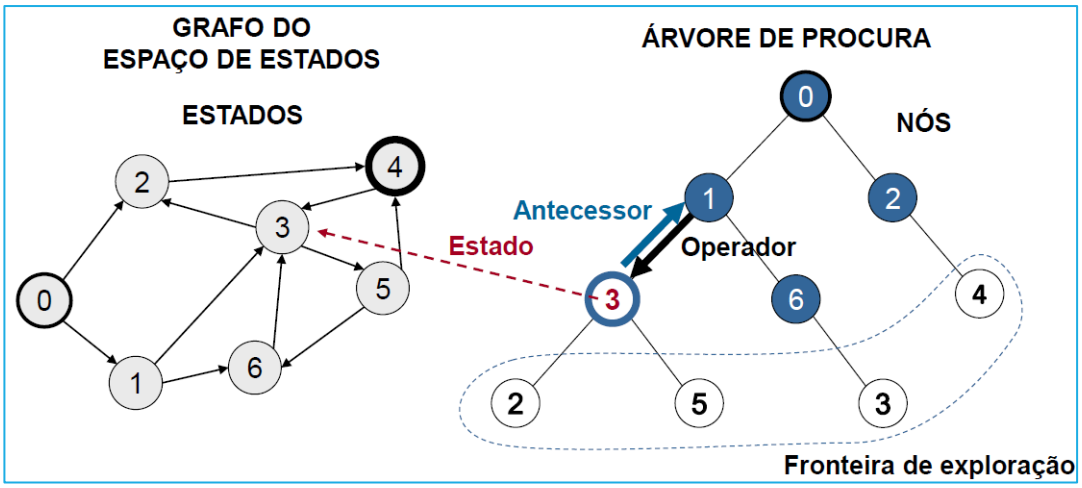

Figura 4 - Grafo de Espaço de Estados e Árvore de Procura

O que processo de todos os mecanismos de procura passa por explorar um Nó, verificar se o Nó é o objetivo, caso não o seja, expande-se o Nó e coloca-se os "filhos" na Fronteira de Exploração, e de seguida vai-se explorar o próximo Nó que está na Fronteira, repetindo o processo até chegar o Nó objetivo. Quando se chegar ao objetivo, gera-se a solução que consiste no caminho, a sequência de Nós até chegar ao objetivo.

Como referido, a ordem que se define para colocar os Nós não explorados na fronteira determina a estratégia de controlo de procura. Assim sendo, estudou-se os seguintes métodos:

2.8.1. Procura em Largura (Breadth-First Search)

Neste método de procura, após se verificar que o Nó onde se encontra não é o objetivo, expande-se o Nó, colocando os "filhos" na última posição da Fronteira de Exploração. O próximo Nó a ser explorado é o mais antigo da Fronteira, ou seja, explora-se todos os "filhos" de um Nó, depois vai-se ao primeiro "filho" e explora-se todos os seus sucessores, depois vai ao segundo "filho" e explora-se os sucessores deste, assim sucessivamente até que o Nó que está a ser explorado seja o objetivo. Com este método tem-se a certeza que se chega ao objetivo, é uma solução completa, e se existir várias soluções, a solução encontrada é a melhor. Porém, ao explorar todos os Nós, vai demorar muito tempo e ocupar mais memória, tendo uma complexidade tando temporal como espacial.

2.8.2. Procura em Profundidade (Depth-First Search)

Contrariamente á Procura em Largura, quando ao explorar um Nó, este não seja o objetivo, ao expandir o Nó, coloca os seus "filhos" na primeira posição da Fronteira de Exploração, explorando primeiro os Nós mais recentes da Fronteira, ou seja, explora-se um "ramo" completo da árvore, e só depois é que se passa para outro, vai-se explorando o "filho" do "filho". Assim caso haja várias soluções para o objetivo, não garante a melhor solução, não sendo um método de procura ótimo.

Este método tem duas variantes, Procura em Profundidade Limitada e a Iterativa. Na Profundidade Limitada, coloca-se uma profundidade máxima de exploração, sendo que a profundidade, é o nível onde se encontra o Nó, incrementando quando se vai descendo na árvore, a partir da raiz. Desta forma, é possível que não se encontre uma solução, não sendo um método nem ótimo nem completo. A Profundidade Iterativa, vai aumentando a profundidade, começando em 1, explorando apenas o primeiro "filho", e vai iterando a profundidade. Assim é possível encontrar até a melhor solução, sendo um método ótimo e completo.

2.8.3. Procura Melhor-Primeiro (Best-First)

Para esta procura é necessário avaliar cada estado, utilizando assim uma função para tal, normalmente a função retorna uma estimativa do custo da solução através do Nó **n** em *f(n)*. Aqui a fronteira é ordenada por ordem crescente da função.

No caso estudado, vai ser uma Procura de Custo Uniforme, onde a estratégia de procura é de explorar os caminhos com menor custo. Este método de procura irá sempre chegar a melhor solução, sendo esta a de menor custo, logo é um método completo e ótimo. A complexidade temporal e espacial, aqui, vai depender do fator de ramificação, do custo da solução ótima e do custo mínimo de uma transição.

2.8.4. Procura em Grafos com Ciclos

Qualquer procura em Grafos pode ter ciclos, múltiplas transições para o mesmo estado. Para evitar o desperdício de recursos, tanto do tempo como de memória, para além de se ter uma fronteira com os Nós ainda não explorados, deve-se ter uma lista dos Nós já explorados, e impedir de serem novamente explorados. Ou seja, para gerar um novo Nó sucessor, expandir, é necessário ter em conta que, se o nó ainda não foi gerado nem expandido, deve-se colocar na fronteira de exploração, caso o Nó já esteja na fronteira, deve se verificar se o Nó foi gerado através de um caminho mais curto, com menor custo, assim sendo, remover o Nó anterior da fronteira e inserir o novo lá. Por fim, caso o nó já tenha sido expandido, e o Nó foi atingido através de um caminho mais curto, remove-se o anterior da lista dos expandidos e coloca-se o novo, com um caminho com menor custo na fronteira.

Os métodos de procura mencionados acima, são métodos de procura **não Informada**, ou seja, o método de exploração vai analisar tudo sem analisar o problema, é uma procura exaustiva, pois não tira partido de conhecimento do domínio do problema para ordenar a fronteira. Em contraste, existe métodos de procura **Informada**, onde a estratégia já vai mais a fundo no problema, vai conhecer o que existe e seleciona o melhor caminho, sendo uma procura guiada, seletiva, tirando partido de conhecimento do domínio do problema para ordenar a fronteira de exploração.

Neste tema, foi apresentado a função heurística *h(n)*, onde representa uma estimativa do custo do percurso a começar pelo Nó *n* acabando no Nó objetivo, utilizando o conhecimento do domínio do problema, guiando assim a procura. A função depende apenas de *n,* e do seu estado e o objetivo, o percurso para chegar ao Nó *n* não é relevante.

No item 2.8.3 explicou-se o método de Procura Melhor-Primeiro, e este tem 3 variantes, sendo que uma delas é a Procura de Custo Uniforma também já mencionada no item, onde não tira partido do conhecimento do domínio do problema a partir da função heurística. Por outro lado, os próximos dois métodos já são de procura Informada.

2.8.5. Procura Sôfrega (Greedy Search)

Neste método não se têm em conta o custo do valor explorado, a função *f(n)*, função que retorna um valor que decidirá qual a posição que o Nó fica na fronteira, é igual a função heurística *h(n)*, assim minimiza o custo local, produzindo apenas soluções sub-ótimas, pois pode não obter a melhor solução.

2.8.6. Procura A*

Com este método, a função *f(n)* já vai ser a soma da função heurística com a soma do custo do Nó inicial até ao Nó *n*. Aqui tem se uma heurística admissível, onde a função vai ser o custo mínimo do Nó *n* até ao objetivo, conseguindo uma solução ótima, a estimativa de custo é sempre inferior ou igual ao custo efetivo mínimo. Para conseguir uma heurística admissível, é necessário retirar restrições associadas ao problema.

Se não se retirar os nós já explorados a heurística pode não ser consistente, ou seja, o custo de transição para o estado seguinte mais a heurística do estado seguinte é maior ou igual a heurística do estado *n*, e a heurística do nó objetivo é 0. Notar que uma heurística consistente é também admissível por aqui o custo da solução ótima é maior ou igual a função *f(n)*, porém a heurística admissível pode não ser consistente ou monótona.

Deste forma consegue-se reduzir a complexidade de procura.

Com uma heurística consistente, para expandir e gerar o Nó sucessor, as condições são iguais ao do item 2.8.4, com a diferença de quando o Nó sucessor já foi explorado, este é eliminado, em vez de ser substituído.

Este método é bastante eficiente, pois sempre que um nó é expandido o percurso desse nó é ótimo, e os que são expandidos a função *f(n)* tem um valor menor que o custo da solução ótima, podendo eventualmente expandir nós com o valor de *f(n)* iguais ao custo final da solução ótima. Com isto, é expandido menos nós. Porém, continua a haver complexidade combinatória, pois o número de nós expandidos continua a ser um valor exponencial relacionado com a dimensão do percurso até ao objetivo. Para além disso, podemos não conseguir produzir soluções em tempo real , e por isso pode trazer problemas em certas situações, sendo assim tem que se abdicar de ter uma solução ótima e utilizar o método de Procura Sôfrega.

Arquitetura de Agentes Deliberativos

Já foi referido os agentes reativos sem estados, onde estes tomam decisões com base no presente, os agentes reativos com estado, tomando decisões com base no passado e no presente, evitando repetir movimentos. Agora tem-se os agentes deliberativos, que tomam decisões com base no passado, presente e no futuro, antecipando.

Para haver tal raciocínio, o agente precisa de simular o ambiente onde se encontra, ter uma representação interna do mundo. Cada opção, cada estado vai precisar de ter um custo associado e uma utilidade. Vai ser preciso ter a representação dos objetivos a atingir, das ações a realizar e como referido do mundo, e assim vai ser possível obter um plano de execução. O agente vai decidir o que fazer, por onde vai, que estados vai transitar e como fazer, um plano de ações para executar, para atingir o objetivo.

O agente vai continuar a percecionar o ambiente, com isso constrói uma representação do modelo do mundo, de seguida vai deliberar o que fazer e de seguida planeia como fazer, executando no final o plano.

Contudo, existe uma questão, o ambiente pode mudar durante o raciocínio, durante a deliberação e planeamento, o que pode tornar o plano invalido, porém, o agente já o vai executar. Por isso, o agente vai ter que reconsiderar, refazer o processo de novo, reavaliar as opções, e se necessário alterar os planos.

Ao raciocínio prático acrescenta se a reconsideração, após a atualizar o modelo do mundo, o agente verifica de houve alterações e se o plano ainda é valido, e caso o mundo altere ou o plano já não for válido, o agente delibera e planeia antes de executar.

Podemos ver a arquitetura do agente deliberativo na *Figura 5* (retirado do documento Arquitetura de Agentes Deliberativos, Luís Morgado página 10).

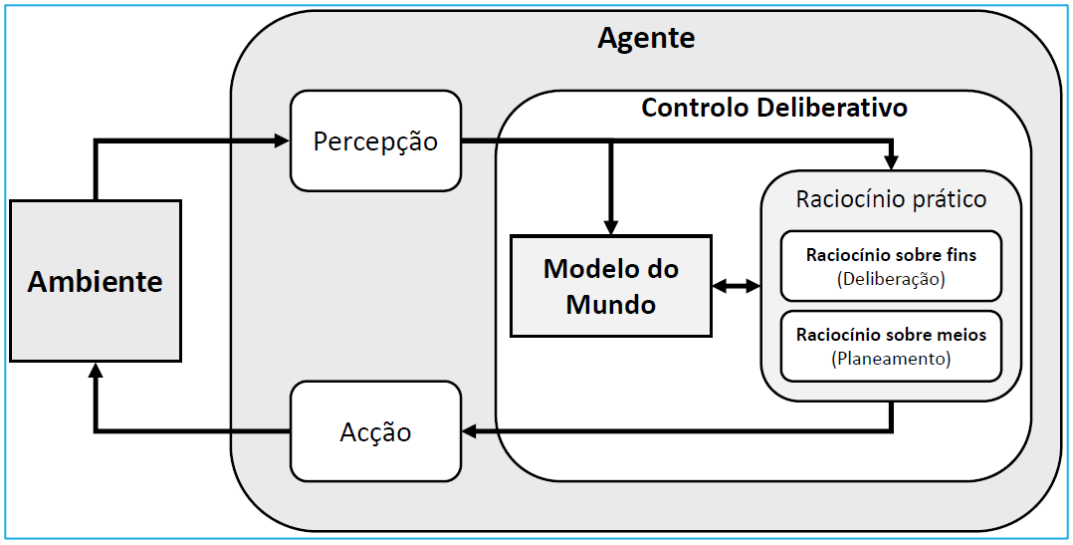

Figura 5 - Arquitetura de Agente Deliberativo

2.9. Planeamento automático

O planeamento automático é o raciocínio automático a fim de planear uma sequência de ações para um agente inteligente executar. Existem diferentes métodos para conseguir um plano de ações tendo como base certos mecanismos de procura, tais como a procura em Espaços de Espatos e processos de decisão de *Markov*.

Existe portando um planeador que recebe os objetivos, o modelo de planeamento e o estado inicial, gerando um plano.

A principal questão quando se fala em processos de decisão, é a de qual a opção que vai dar mais benefício a longo prazo. Eis aqui o problema da decisão ao longo do tempo, a utilidade de uma ação depende de uma sequência de decisões, onde cada decisão pode dar tanto ganhos como perdas, sendo que se pode acumular.

Uma cadeia de *Markov* é um modelo estocástico onde a probabilidade de cada estado seguinte depende apenas do estado presente. Na representação do mundo sob a forma de Processos de Decisão de *Markov*, tem-se o conjunto de estados do mundo, o conjunto de ações possíveis em cada estado, a probabilidade de transição de um estado para outro através de uma ação e a sua recompensa esperada, uma taxa de desconto para recompensas diferidas no tempo, normalmente representado como gama, sendo que quanto maior o valor, mais o futuro vai ser explorado, o que é ideal para obter decisões mais acertadas. Este desconto é feito quando se está a calcular a utilidade de uma ação, onde no valor da utilidade do estado seguinte é aplicado o desconto. A utilidade vai ter um valor cumulativo de recompensas de estado para estado, com a influência de um valor gama, e o agente vai pelo caminho mais benéfico, com uma melhor recompensa, sendo que se o valor do caminho for negativo, significa que o agente vai ficar a perder e por isso, a melhor decisão é ficar parado.

Para representar o comportamento do agente, tem-se as políticas comportamentais que definem a ação que o agente deve escolher e realizar para cada estado. Para o cálculo da política tem que se verificar para cada estado qual a ação com o valor máximo, com uma melhor utilidade.

A diferença entre o planeador com base na procura em espaço de estados e com base em processos de decisão de *Markov*, é que enquanto o planeador da procura de espaço de estados retorna um plano de ações, o planeador com base em processos de decisão de *Markov* retorna uma política, porém, não deixa de ser ações a ser executadas que são retornadas.

2.10. Aprendizagem Por Reforço

A aprendizagem é o método de, com base em experiência, de conhecimento e de desempenho, aumentar e melhorar as capacidades de fazer algo. Tendo dois tipos de aprendizagem, a conceptual e a comportamental, tendo a conceptual como objetivo, aprender o conceito, e a comportamental aprender o que fazer, que comportamentos ter, através de reforço.

Na aprendizagem por reforço, o agente precisa de interagir com o ambiente, passar pelos estados, executar as ações e ver se fica a ganhar ou a perder, sabendo assim se deve voltar a fazer essa ação ou não.

Então, para obter um percurso para chegar ao objetivo é necessário explorar as diferentes opções, aprender por tanto. Contudo, o agente vai ter que aproveitar o que aprender, vai ter que escolher uma ação que leva à melhor recompensa, a chamada ação Sôfrega (*Greedy*).

Existem duas estratégias de seleção de ação, a estratégia *greedy*, que escolhe a ação que leva a recompensa máxima, e a ação *ε-greedy*, ação que vai depender de um valor *épsilon*, vai tanto explorar como selecionar a ação que maximiza o valor de Q, valor de num estado realizar uma ação.

Para calcular o valor, estimar o valor Estado-Ação, do estado observado e da ação realizada, vai ser o valor do Estado-Ação do estado seguinte, multiplicado por um fator, adicionando um reforço, sendo este reforço os ganhos ou as perdas.

Para ambientes não estacionários, é necessário subtrair a estimativa atual de Q com a estimativa anterior, multiplicando por um alfa para no fim somar com o valor de Q atual, atualizando assim a estimativa do valor com base na mudança entre instantes sucessivos.

Desta forma pode se aplicar o Algoritmo *SARSA* onde se inicializa o estado, escolhe-se a ação de acordo com o estado através da política de Q, como o *ε-greedy*, executa se a ação escolhida, observando o reforço e o estado seguinte, de seguida escolhe-se a ação seguinte de acordo com o estado seguinte com base na política derivada de Q, atualizando o Estado-Ação com a diferença temporal falada no parágrafo anterior. E repete-se o processo até que o estado onde se encontra seja um estado terminal.

No Algoritmo *SARSA*, a estimativa de retorno, valor estimado de realizar a ação *a* num estado *s*, é considerada a ação seguinte, a ação de estado seguinte para o estado depois desse. No Algoritmo *Q-Learning*, considera-se a ação seguinte correspondente a melhor estimativa do valor Estado-Ação do estado seguinte, ou seja, verifica, de entre todas as ações possíveis no estado seguinte, qual a melhor estimativa de retorno.

Existem dois tipos de aprendizagem, *On-policy* e *Off-policy*. Por um lado, na *On-policy*, política de seleção de ação única, é usada a mesma política de seleção tanto para o comportamento como para a propagação de valor, onde é utilizado a por exemplo, a política *ε-greedy*, para a exploração de todas as ações. Por outro lado, na *Off-policy*, políticas de seleção de ação diferenciadas, é utilizada a mesma política de seleção de ação também para o comportamento e para a propagação de valor, e tem a otimização da função valor Estado-Ação (Q(s,a)).

Em suma, para se convergir para o valor ótimo, não se pode não só explorar como se tem que aproveitar o que foi explorado e aproveitar as recompensas, não sendo também a única ação a fazer. Contudo, não se pode parar de explorar, pode haver sempre uma solução mais benéfica que só explorando é que saberá, sendo que se deve reduzir a exploração de forma progressiva.

3. Projeto Realizado

3.1. Parte 1 + Parte 2 – Jogo Fotografar os animais.

3.1.1. Introdução

Pretende-se implementar um jogo com uma personagem virtual que interage com um jogador humano.

O jogo consiste num ambiente onde a personagem tem por objetivo registar a presença de animais através de fotografias.

Quando o jogo se inicia a personagem fica a procurar animais. Quando deteta algum ruído aproxima-se e fica a inspecionar a zona, procurando a fonte do ruído. Quando volta a haver silêncio a personagem continua a procurar animais.

Quando deteta um animal a personagem aproxima-se e fica a observar o animal, preparando-se para o registar. Na situação de registo, se o animal fugir a personagem fica a inspecionar a zona, à procura de uma fonte de ruído. Caso o animal continue, a personagem faz o registo do animal fotografando-o. Na situação de registo, caso ocorra a fuga do animal ou a personagem tenha conseguido uma fotografia do animal, esta fica novamente à procura.

A interação com o jogador é realizada em modo de texto.

(Retirado do documento Projecto -Parte 1, Luís Morgado página 1)

O jogo decorre num ambiente virtual que contém vários animais. Existe uma personagem que vai estar a registar estes animais fotografando-os.

Podemos decompor o problema em 3 componentes: Jogo, Ambiente e Personagem.

Para esta parte, executou-se no *Java Virtual Machine*. A versão do java utilizada foi o *jdk* 18.

Para a realização desta parte, implementou se a arquitetura que vemos na *Figura 6* (figura retirada do documento Projecto -Parte 1, Luís Morgado página 13), e implementou se uma máquina de estados, com uma classe que representa a mesma , que contém o Estado atual, sendo o Estado um objeto que pode transitar para outro Estado, dependendo do Evento, gerando uma ação, sendo que cada estado terá várias transições, dependendo dos eventos e ações que existem. Uma Transição, contém o Estado sucessor e a ação.

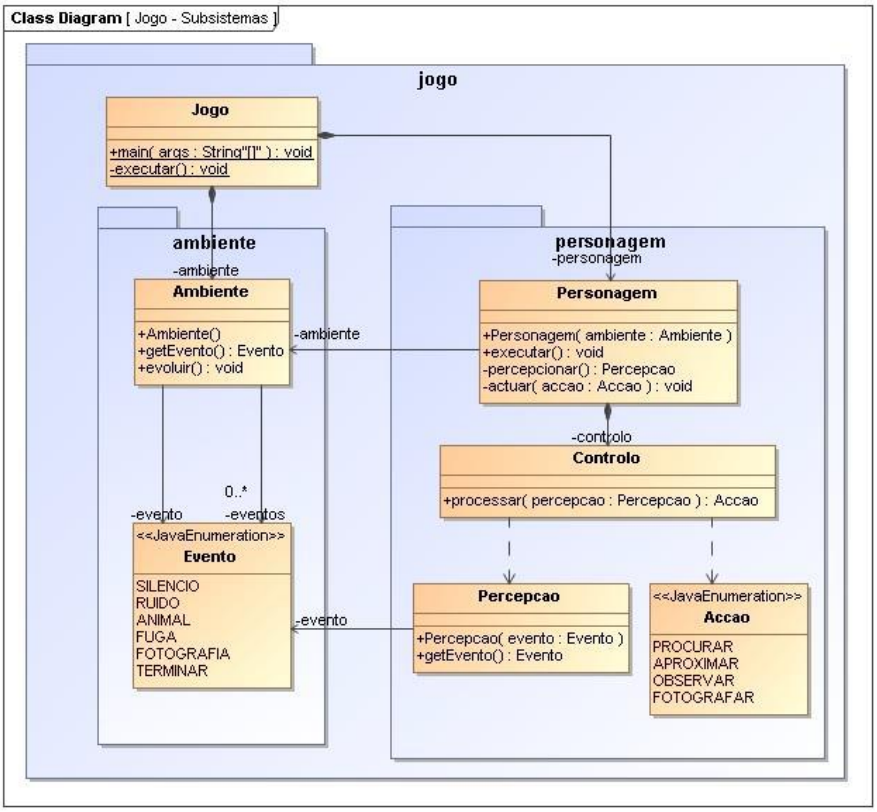

Figura 6 - Diagrama de Classes – Jogo

Podemos retirar a classe *Jogo* onde tem o método *main()*, que irá executar o jogo. Após correr o programa, vai ser criado uma intância da classe *Personagem* e da classe *Ambiente* (ambas as casses compõem a classe *Jogo*). É chamada depois o método *executar()* que vai executar a personagem, e vai evoluir o ambiente, e vai repetir até que o utilizador selecionar o evento *TERMINAR*.

A classe *Ambiente* vai representar um ambiente virtual que contém animais. Este ambiente contém vários eventos, os quais estão no enumerado Evento. No decorrer do jogo, o utilizador vai escolher um dos eventos onde vai fazer com que a personagem tenha uma certa ação, sendo este o propósito do método *evoluir()*, mudar de eventos.

A personagem vai seguir um modelo, modelo que, a partir dos sensores vai percecionar o ambiente, processar o que captou nos sensores, e atuar através de atuadores. Neste caso, a personagem não tem sensores, simplesmente utiliza o método *getEvento()* da classe *Ambiente* para saber percepcionar. Neste jogo, a personagem vai ver em que evento é que o ambiente se econtra, processa o mesmo e escolhe uma ação dependendo do evento, e por fim atua. Para tal, é criado o método percecionar que cria uma instancia de Percepcao, one guardará o evento em que o ambiente se encontra. Para se processar, é criada uma classe *Controlo*, classe com a função de processar o evento e decidir qual a ação a atuar, assim, a classe *Personagem*, só tem que ver qual é o evento do ambiente que a classe *Controlo* trata do resto. A classe *Controlo*, como já referido, apartir do evento percecionado, vai decidir que ação a personagem vai atuar. Para tal, existe um enumerado de ações (*Accao*). O Controlo vai criar para cada ação uma transição que gerará um novo estado, e será uma transição para cada Evento que possa surgir.

Em suma, a Personagem vai ter a Perceção de um Evento do Ambiente, com a ajuda do Controlo, a Personagem vai processar a perceção, e apartir de uma Máquina de Estados, vai fazer a Transição do Estado atual para o Estado Sucessor, gerando uma ação que a personagem vai por fim atuar.

3.2. Parte 3 + Parte 4 + Parte 5

Como foi falado no item 2.6 um agente com uma arquitetura reativa vai percecionar o ambiente, vai ter um estímulo, tendo uma resposta que provoca uma ação. Nesta parte do projeto elaborou-se uma biblioteca denominada de *ecr*, nesta fez se classes relacionadas com alguns pontos falados no item 2.6, como o estímulo, resposta, reação, comportamento, etc.

Para implementar esta parte, seguimos as práticas da engenharia de software, começando por observar diagramas de classes e diagramas de sequências, realizamos o código estrutural e de seguida o comportamental.

Para haver uma reação, vai ser necessário um estímulo e depois é provocada uma resposta contendo esta uma ação. Fica se, portanto, com o diagrame de sequência da *Figura 7* (figura retirada do documento Projecto -Parte 4, Luís Morgado página 1) . Vai ser ativada uma reação, deteta-se um estímulo através da perceção, ficando com um valor de intensidade e dependendo desse valor, e da perceção, vai se ativar uma ação, retornando a mesma.

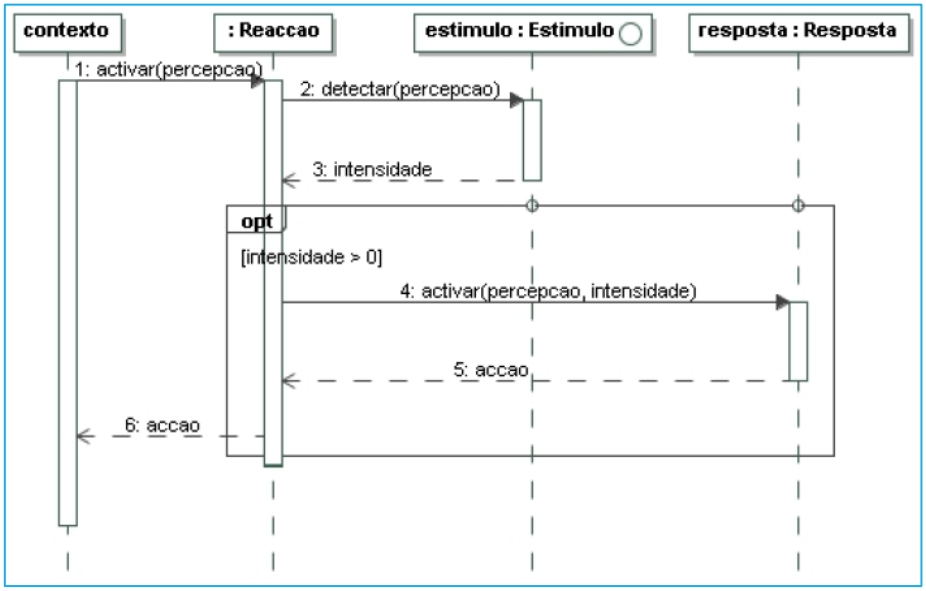

Figura 7 - Diagrama de sequência da reação de um agente

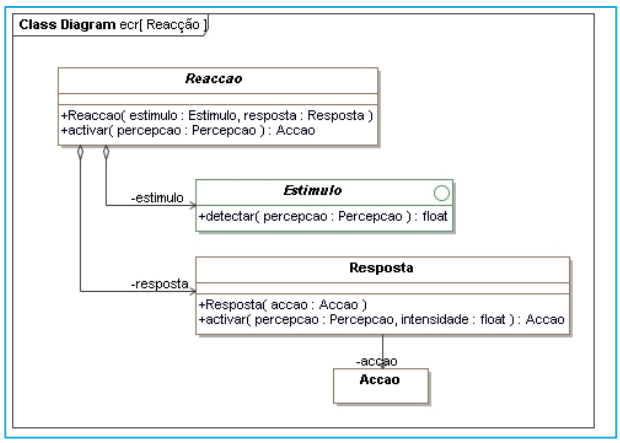

Figura 8 - Diagrama de classes da Reação

Obtemos agora o diagrama de classes da reação, como vemos na *Figura 8* (figura retirada do documento Projecto -Parte 4, Luís Morgado página 2). Existe uma classe *Reaccao* que é composta por um *Estimulo*, e uma Resposta que por sua vez contém uma classe *Accao*.

A classe *Reaccao* vai implementar a interface *Comportamento*, o que faz sentido pois um comportamento relaciona os padrões de perceção com os padrões de ação, assim como a classe *Reaccao*. Esta classe contém um só método, único método presente na interface do *Comportamento*, que se observa na *Listagem 1*, sendo que este recebe a perceção, produz um valor da intensidade a partir de uma classe que implementa a interface *Estimulo*, e com esse valor e a perceção que se percecionou, consegue-se gerar uma resposta produzindo uma ação.

```
def activar(self, percepcao):
      intensidade = self._estimulo.detectar(percepcao)
      if intensidade > 0:
          accao = self._resposta.activar(percepcao, intensidade)
          return accao
```

Listagem 1 - Função activar da classe Reaccao

É criada então uma classe que implementa a interface *Comportamento*, denominada de *ComportComp*, ou seja, comportamento composto, e como já referido no item 2.6 o comportamento composto é um conjunto de comportamentos. Logo, para implementar esta classe é necessário ter uma lista de comportamentos. Como estende a classe *Comportamento*, é implementado a função *activar()*, onde percorre-se a lista de comportamentos e chama se o método *activar()* de cada um da lista, ficando com uma lista de ações, pois o método retorna uma ação.

Depois de percecionar algo, depois do estímulo, vai se obter uma resposta e produzir uma ação. Porém temos uma lista de ações, qual escolher e como escolher?

Como já foi mencionado, existem várias formas de o fazer. Foi implementado duas, a hierarquia e a prioridade. A classe *ComportComp* tem a função *selecionar_accao()* abstrata, e as classes *Hierarquia* e *Prioridade*, vão estender o *ComportComp*, decidindo elas como vão selecionara a ação. Por um lado, se se optar pela escolha da ação por hierarquia, no método de *selecionar_accao()*, vai retornar se a ação que está na primeira posição da lista. Por outro lado, escolhendo pela prioridade, o método vai retornar a ação da lista que tem maior prioridade, prioridade esta que é definida na *Reaccao*, é calculada uma intensidade pelo método do *Estimulo*, e na classe *Resposta*, ao ativar, é declarado que a prioridade da ação (inicializado no construtor da classe) é igual a intensidade calculada no estímulo.

Para agora implementar o Agente Reativo, falta criar um Controlo Reativo e reações.

Com o intuito de ajudar na execução do projeto e visualizar os resultados, é utilizado uma biblioteca denominada de Simulador de Ambiente de Execução (SAE, *sae* como esta identificada no projeto). Aqui existem interfaces e métodos abstratos para aplicar no agente que se esta a desenvolver, no ambiente criado e no final é possível visualizar o agente a executar e simular o código desenvolvido.

Assim sendo, foi implementada uma classe *ControloReact*, que estende *Controlo* de *sae*. Esta classe é composta por uma classe que implemente *Comportamento*, como por exemplo a classe do comportamento composto(*ComportComp*), e contém apenas o método *processar()*, que recebendo uma perceção vai ativar o/os comportamento/comportamentos.

Para o projeto, o cenário é o seguinte: Existe um ambiente que tem obstáculos e alvos, existe um agente que tem o objetivo de recolher os alvos. Para tal, é usado o Controlo Reativo do parágrafo anterior.

Para concluir o controlo reativo, falta as reações. Para este caso implementou se três, o explorar, evitar e aproximar.

A classe *Explorar* criada implementa a interface *Comportamento*, sendo que este comportamento, esta reação consiste em o agente mover se numa direção aleatória, explorando o ambiente. A classe depende então das direções, *NORTE, SUL, ESTE, OESTE,* que se podem encontrar na biblioteca *sae*, e depende de uma resposta. Quando foi criada a biblioteca *ecr*, foi criada uma classe *Resposta*, onde no seu construtor recebe uma ação e guarda. Para criar a resposta desta ação, foi implementada uma classe que estende a classe *Resposta*, *RespostaMover*. A classe quando instanciada receber uma direção, e cria uma ação da classe *Accao* do módulo *sae*, onde necessita dessa direção. Quando o comportamento de explorar é ativado, vai ser escolhida aleatoriamente uma direção, de seguida é criado uma instância do *RespostaMover*, com a direção gerada, por fim é ativada a resposta, que é o retornar da ação, neste caso a ação de mover para a direção escolhida.

A reação de aproximar, é a ação do agente se mover para se aproximar de um alvo. Para esta implementação, é criada a classe *AproximarDir,* que estende *Reaccao*. Para criar uma nova instância de uma reação é necessária uma instância de *Estimulo* e de *Resposta*, assim foi criada a classe *EstimuloAlvo*, que necessita de uma direção (esta fornecida ao criar a instância de *AproximarDir*), e é instanciada a classe *RespostaMover* (recebendo a mesma distancia). Quando se ativa a reação, ao detetar o estímulo, vai se buscar que elemento foi percecionado na direção recebida, e qual a distancia a que se encontra, e se o elemento for um alvo (identificando com a ajuda da *sae*), é retornado o valor da intensidade da perceção, que neste caso, quanto mais perto do alvo, mais intenso é a perceção, pois, o objetivo é o agente chegar ao alvo, se ele percecionar vários alvos ao mesmo tempo ele vai ao mais perto, o que tiver um valor de intensidade maior. Como podemos ver, aqui vai ser usado um método de escolha de ação por prioridade, desta forma é criada uma classe *AproximarAlvo* que estende *Prioridade*, e a lista de comportamentos que recebe é a da reação *AproximarDir* para todas as direções. Quando se ativar a resposta, vai ser um *RespostaMover* e a ação que vai ser retornada, com o selecionar ação de ser a prioritária, é a ação que faz com o agente se mova para o alvo mais próximo.

Por fim, o agente pode evitar obstáculos, para tal, é criada a classe *EvitarDir*, sendo esta uma *Reaccao*, necessita também de um parâmetro *Estimulo* e *Resposta*. Para o estímulo é implementada a classe *EstimuloObst*, onde ao detetar, a partir da perceção na especifica direção, sabe o elemento que se encontra nessa direção, e caso este elemento seja um *OBSTACULO* (segundo *sae*), e o agente esteja em contacto com o mesmo é retornada uma intensidade de 1, o máximo. A resposta é passada quando se instância a classe *EvitarDir*, esta resposta é de uma classe nova, *RespostaEvitar*, onde estende a classe *RespostaMover*. No construtor, é criada uma lista de direções e chama se o *super()* passando uma direção inicial predefinida(*ESTE* neste caso). Quando se ativa a resposta, para obter uma ação, é percecionado se o agente esta em contacto com algum obstáculo na direção inicial, e se for o caso é procurado uma ação livre, onde vai a uma lista de direções onde o agente não está em contacto com obstáculo, escolher aleatoriamente, é alterado a direção da ação criada inicialmente, e recursivamente, ativa se a resposta com a nova ação, desta vez, como a condição de estar em contacto com obstáculo já vai ser falsa, é ativada a ação do *super*, do *RespostaMover*.

Na *Figura 9* (figura retirada do documento Projecto -Parte 5, Luís Morgado página 5) podemos ver o diagrama de atividades do ativar da *RespostaEvitar*, que auxiliou na implementação do algoritmo. Desta forma, a escolha da ação para este caso é a Hierarquia, por consequente criou se a classe *EvitarObst*, que estende *Hierarquia*, onde é instanciada a *RespostaEvitar*, sendo a lista de comportamentos a reação *EvitarDir* para todas as direções.

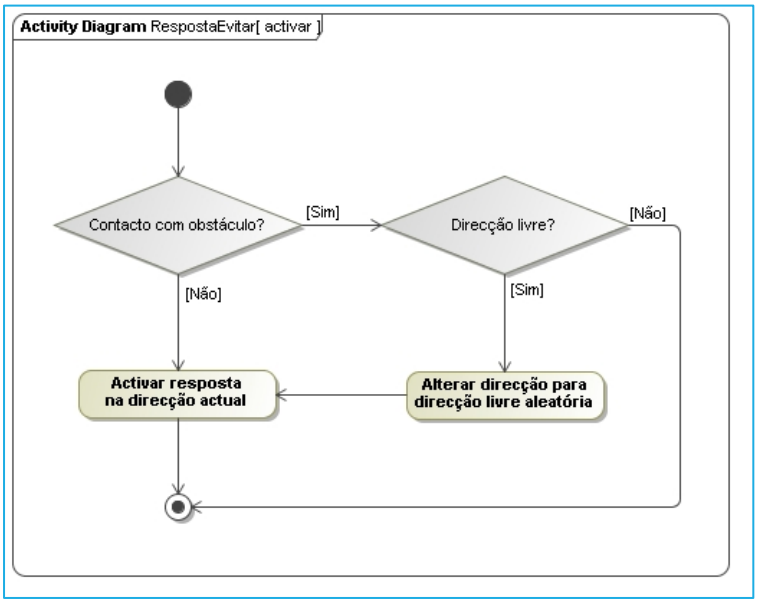

Figura 9 - Diagrama de Atividade do método activar()

Por fim, para a escolha de que ação vai ser executada, foi criada uma classe *Recolher* que é um comportamento composto que escolhe o comportamento pela hierarquia, e a lista dos comportamentos, pela ordem da hierarquia são comportamento de aproximar um alvo (*classe AproximarAlvo* foi adicionada na lista), caso não possa aproximar alvo tem o comportamento de evitar obstáculo (classe *EvitarObst* foi adicionado na lista), e se não houver obstáculo o agente explora (classe *Explorar* adicionado na lista).

Para terminar a simulação de um agente reativo, foi criado um módulo *teste_react.py* que implementa esta arquitetura. Para tal bastou instanciar o controlo, *ControloReact*, passando como comportamento uma nova instância do comportamento *Recolher* que contêm os comportamentos todos. Por fim, utilizando o simulador da biblioteca *sae*, chama se o método *Simulador.executar()*. Este simulador tem vários ambientes de diferentes dimensões, onde é passado o número do ambiente pretendido com argumento, neste caso simulou-se no 1, e também como argumento vai o controlo. O resultado é o que aparece nas figuras seguintes, sendo que do lado esquerdo é o ambiente, e no direito mostras as perceções do agente em cada direção. A verde apresenta-se os alvos, cinzento os obstáculos, e a amarelo, e quadrado com borda amarela o agente.

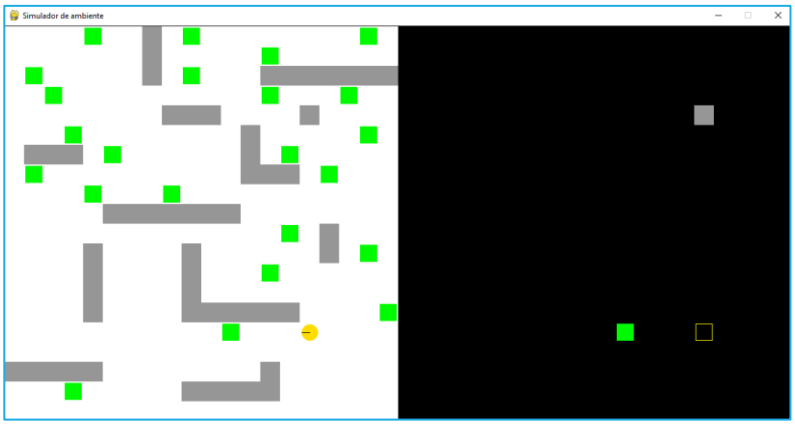

Figura 10 - Simulador teste_react P1

Na *Figura 10* confirmamos que o agente perceciona o alvo a *ESTE*, e um obstáculo a *NORTE*. A decisão é de aproximar-se do alvo.

Figura 11 - Simulador teste_react P2

Na *Figura 11* observamos que o agente perceciona dois alvos, porém aproximar-se-á do que está na direção *NORTE*, pois encontra se mais perto, logo tem uma maior prioridade.

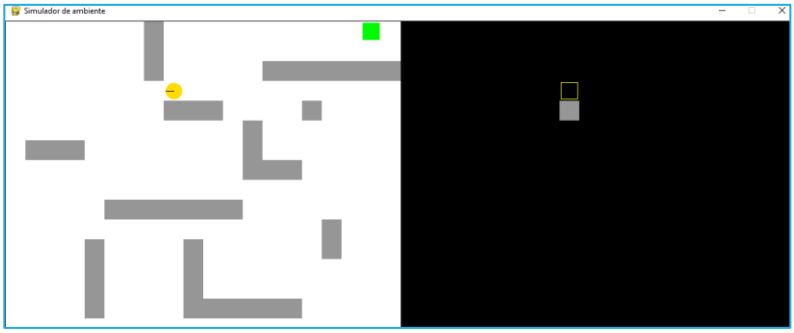

Figura 12 - Simulador teste_react P3

Na *Figura 12* o agente perceciona um alvo a *SUL*, não havendo alvos percecionados, o agente vai evitar o obstáculo. Como está predefenido, o agente verifica primeiro se na direção a *ESTE* o agente está em contancto com algum obstáculo, não sendo este o caso, o agente tem a ação de se mover nessa direção.

3.3. Parte 6 + Parte 7 + Parte 8

No item 2.8 foi estudado mecanismos de procura em espaços de estados, por isso implementou-se um agente que consiga recolher alvos utilizando os diferentes mecanismos.

Para começar criou se uma biblioteca com classes relacionadas com o Modelo de um problema, a biblioteca *mod*. Aqui existe a classe *Estado*, que contem um id que é único, sendo que o modo que definido os valores depende dos cenários e dos tipos de estados. Contém também a interface *Operador* onde as classes que implementarem-na têm que definir como vão aplicar o operador a um estado recebido como argumento, e têm de definir o cálculo do custo da mudança de um estado para o estado sucessor. Por fim foi adicionado, dentro de um *package* a classe *Problema* onde este vai ter guardado o estado inicial e a lista de operadores, inicializados no construtor da classe. As classes que estenderem esta, vão definir qual é o objetivo do problema e têm que definir a função que verifica se o estado que recebe como argumento corresponde ao estado final, objetivo.

Como temos vários mecanismos de procura, sendo que têm certas partes em comum, aplicando as boas praticas de engenharia de software, criou-se a classe *MecanismoProcura*, *Fronteira*, *No* e *Solucao*.

Um Mecanismo de Procura, vai conter uma fronteira onde vai colocar os Nós explorados. Todos os mecanismos vão seguir mais ou menos o mesmo algoritmo para encontrar a solução do problema, por isso existe uma classe chamada *resolver()* que vai ter o código da Listagem 2, aqui é criado um objeto da classe *No*(falado no seguinte parágrafo), cujo o estado que representa é o estado inicial do problema, de seguida é inicializada a fronteira, e aplica se o método memorizar, onde no geral vai colocar o nó que recebe na fronteira, porém cada mecanismo de procura vai implementar esse método. Após colocar se o Nó inicial na fronteira, começa se um *loop* que percorre a fronteira, os nos ainda não explorados mas já expandidos, e verifica se o Nó que remove é o objetivo, e se for o caso, constrói se a solução a partir da classe *Solucao*, caso contrário, expande se o Nó, onde basicamente percorre-se a lista de operadores e aplica-se o operador, recebendo um estado seguinte caso exista, guardando o Nó gerado, com o estado, e aplica-se o método *memorizar()*, para colocar esse estado na fronteira para ser explorado. No método expandir, como se vai estar a retornar possivelmente vários estados, usa-se o método *yield* que vai retornando um nó de cada vez, assim temos código mais simplificado.

```
def resolver(self, problema):
        no = No(problema.estado_inicial)
        self._fronteira = self.iniciar_fronteira()
        self.memorizar(no)
        while not self._fronteira.vazia():
            no = self._fronteira.remover()
            if problema.objectivo(no.estado):
                return Solucao(no)
            self._nosProcessados += 1
            for child in self.expandir(problema, no):
                self.memorizar(child)
        return None
```

Listagem 2 - método resolver da classe MecanismoProcura

A classe *No*, representa um Nó, sendo que terá certos atributos relacionados, como a profundidade, em que nível se encontra na árvore, o custo, a soma do custo do nó antecessor com o custo do operador do Nó antecessor para este, o operador e o Nó antecessor.

A diferença entre os mecanismos de procura encontra-se na fronteira, mais precisamente a ordem que esta é ordenada. Por isso, implementou-se uma classe *Fronteira*, onde quando é inicializada, inicializa uma lista, contém um método que cerífica se a lista esta vazia, o método *remover()* que remove o primeiro elemento da lista usando o método *pop(0)*, e por fim a classe que vai distinguir as fronteiras, o método *inserir()*, sendo este abstrato para tal acontecer.

Foram então criadas as fronteiras *FronteiraLIFO*, *Last-in-First-Out*, onde se coloca o Nó expandido no princípio da lista para ser o primeiro a ser removido, utilizando assim esta fronteira no mecanismo de procura em profundidade. A *FronteiraFIFO*, *First-in-First-out*, onde se coloca o Nó expandido no final da lista para ser explorado depois, ou seja, é explorado o Nó que está há mais tempo na lista, usado na procura em largura. Por fim, a fronteira prioridade, que vai ordenar numa forma específica definida por um *Avaliador*, esta usada por mecanismos de procura melhor primeiro. Para a *FronteiraLIFO*, basta aplicar o método *insert()* a lista dos nós, inserindo o nó recebido na *index* 0, Enquanto na *FronteiraFIFO*, usa-se o método *append()*. Já na outra fronteira, *FronteiraPrioridade*, é necessário inicializar o avaliador, verificar a prioridade do nó a inserir, e através da função *heappush*(), onde se coloca a lista a adicionar no primeiro argumento e depois, no segundo argumento, colocamos um tuplo com a prioridade do nó, e o nó, desta forma, a função coloca logo o nó na lista no local correto dependendo da prioridade. Desta forma, o remover nesta fronteira, usa-se a função *heappop()* que retira o nó com a menor prioridade.

Quando um mecanismo encontra uma solução, é criada uma nova instância da classe *Solucao*, dando como argumento o nó objetivo. No construtor é guardado o no final, cria-se a lista do percurso e coloca se na posição 0 o nó final. Como cada nó tem o nó antecessor guardado, vai se inserindo na posição 0 o nó antecessor, alterando o nó que se está a ver o anterior para o antecessor, ficando com o *loop* da *Listagem 3* que termina quando um nó não tenha um antecessor, ou seja o nó raiz. Utilizando o *insert()* é colocado o nó no primeiro *index* e os outros que estão já na lista , incrementam um valor no *index*.

```
no = no_final
while no.antecessor:
    self._percurso.insert(0, no.antecessor)
    no = no.antecessor
```

Listagem 3 - criação da lista do percurso

Com a classe *Solucao*, é possível saber a dimensão do percurso, aplicando o método *len()* com a lista do percurso em argumento, é possível saber o custo total, que se vai buscar o custo ao último nó. Também se pode remover um "passo" do percurso, sendo que se devolve uma instância da classe *PassoSolucao()* onde é necessário o estado do primeiro nó e do operador seguinte. Por fim pode se iterar pelo percurso, com o método __iter__, e ir buscar um elemento a lista do percurso. Nestas últimas funções, quando se chama *nomeDaInstanciaDeSolucao.iter()*, é retornando o próximo elemento da lista do percurso, assim como se se chamar com *.getitem()* com o número do mesmo, é recebido o elemento no *index* pedido da lista do percurso.

Agora, com estas classes criadas, pode se implementar cada mecanismo de procura. Primeiro, podemos dividir em duas vertentes, a procura em profundidade e a procura em grafos.

3.3.1. Procura em Profundidade

Para este método de procura foi criada a classe *ProcuraProf()*, esta estende a classe *MecanismoProcura()*. Quando se faz resolver deste mecanismo, inicia a fronteira do tipo LIFO, *Last-in-First-Out*, e o método não sofre alterações, porém, quando se chama o método *memorizar()* passando o Nó a guardar na fronteira este chama o método *inserir()* da fronteira criada.

Para este mecanismo, foi implementado as duas restantes variantes, a Procura Profundidade Limitada e a Iterada, *ProcuraProfLim()* e *ProcuraProfIter()*. Na da procura limitada, esta estende a classe *ProcuraProf()* e quando é chamado o método para resolver, primeiro é guardada a profundidade máxima que se pode pesquisar e depois executa-se o método resolver da *super* classe. No expandir verifica se a profundidade do nó a ser expandido se é maior que a profundidade máxima, se não for o caso, ainda verifica se o nó já foi expandido previamente, vendo os Nós antecessores, e se não estiver repetido executa o método expandir da *super* classe. Devido ao facto de se ter acoplado estas classes, a classe *ProcuraProfIter*, que estende a *ProcuraProfLim*, só terá que se fazer um ajuste no método resolver. Como aqui é para ir resolvendo com diferentes máximos de profundidades, vai ser fazer o método resolver da *super* classe com a profundidade a ir incrementando até que se chegue a uma solução, ou que se alcance a profundidade máxima.

3.3.2. Procura Grafo

O mecanismo de procura em largura e melhor primeiro ainda têm partes em comum, por isso, é criada a classe *ProcuraGrafo()* que estende *MecanisnoProcura()*, e aqui, é criado um dicionário para colocar os nós explorados (sendo o estado do nó a *key* do dicionário), e quando no método resolver chamar o método *memorizar()*, verifica-se se o nó é para memorizar, colocar na fronteira, que só é possível se este ainda não pertence ao dicionário dos nós explorados. Caso for para adicionar a fronteira, coloca se no dicionário o nó.

3.3.3. Procura em Largura

Para este mecanismo, é criada a classe *ProcuraLarg()* que estende a classe *ProcuraGrafo()* e o único método a implementar é o *iniciar_fronteira()*, onde é iniciada uma fronteira to tipo FIFO, *First-In-First-Out*.

3.3.4. Procura Melhor Primeiro

Para este mecanismo, é criada uma classe abstrata, *ProcuraMelhorPrim()* que inicializa a fronteira *FronteiraPrioridade()* onde passa um novo avaliador que é retornado de o método *iniciar_avaliador()* que é método abstrato. Para além disso, adiciona uma condição para o método manter, onde , caso o nó já esteja no dicionário dos explorados, verifica se o custo do nó é menor do que o do nó que está no dicionário, e se sim, guarda o de menor custo.

Para implementar esta classe, usa se o mecanismo de Procura Custo Uniforme, onde inicializa o avaliador onde a prioridade corresponde ao custo do nó, *AvalCustoInif()*.

Como falado nos itens 2.8.5 e 2.8.6, existem as variantes que entram no grupo de procura informada.

3.3.4.1. Procura Informada

Criou se então uma classe abstrata que estende a classe *ProcuraMelhorPrim()*, chamada de *ProcuraInformada()*, onde, antes de chamar o método resolver da *super* classe no ser método de resolver, inicializa uma função heurística, que corresponde a uma interface *Heuristica()*. Esta interface heurística contém um método que representa a função h(), que por sua vez, como já referido, represente uma estimativa do custo do percurso desde um nó até ao objetivo, refletindo conhecimento acerca do domínio do problema para poder guiar a procura.

Para conjugar o método presente na interface *Avaliador* com uma função heurística, é criada a classe abstrata *AvalHeur()* que estende a interface do avaliador e inicializa uma classe heurística e guardada.

3.3.4.1.1. Procura A*

A partir do avaliador acima, podemos criar um avaliador para este método de pesquisa onde na sua classe, *ProcuraAA()*, estende a classe *ProcuraInformada()* e no método *iniciar_avaliador()*, inicia um avaliador chamado *AvalAA()*. Neste avaliador, no método *prioridade()* resolve a função heurística e adiciona o custo do nó.

3.3.4.1.2. Procura Sofrega

De forma semelhante ao mecanismo anterior, criou se uma classe, *ProcuraSofrega()*, que estende a *ProcuraInformada()* e inicializa o avaliador *AvalSofrega()*, sendo que neste, ao contrário do *AvalAA()*, retorna só o valor da função heurística no nó recebido.

Para fazer a simulação de procura em espaços de estado, é preciso ter um problema e todas as suas componentes, ou seja, estados e operadores, sendo também preciso um planeador de trajeto.

Foi criado então a classe *ProblemaPlanTraj()*, que vai ser o problema da simulação, e por isso vai estender a classe *Problema()*. O tipo de estado deste problema vai ser localidades, ou seja, cada nó que tenha um *EstadoLocalidade()* (nome da classe que estende *Estado()*), vai ter uma localidade associada, sendo que o id do estado vai ser um *hash* da *string* que representa a localidade. Faltando agora definir a classe dos operadores, foi implementado o *OperadorLigacao()*, que estende a classe *Operador()*, sendo que ao criar uma instância, é necessário fornecer como argumento a localidade de origem, a de destino e o custo, criando os estados localidades de origem e destino ainda no construtor. Para guardar estas informações para criar um operador, criou-se uma *dataclass*, *Ligacao*, que tem uma *string* com a origem da ligação e outra com o destino e o custo para da transação. Voltando ao *ProblemaPlanTraj()*, ao criar uma instância tem que se passar como argumento uma lista com as ligações, a localidade inicial e a final, sendo que no construtor é criado o *EstadoLocalidade()* para ambas as localidades, e é criada uma lista de operadores a partir da lista com *Ligacao()*, sendo que no método *objetivo()* verifica-se se o estado recebido é igual ao do estado final.

Agora só falta algo que faça o planeamento. Para isso foi criada a classe *PlaneadorTrajecto()*. Esta classe contém o método *planear()* que receber as ligações, e as localidades inicial e final. Cria então o problema, e o *user* tem que escolher o método de procura, que depois é instanciado e executado o método *resolver()* retornando uma solução, caso exista. A classe ainda tem o método para mostrar o trajeto da solução, onde vai iterando cada nó e faz *print* da *string* da localidade e depois o custo final. Ainda é possível verificar as complexidades, a temporal, que é o número de nós processados, o número de nós expandidos e a complexidade espacial que é o número de nós em memória, ou seja, os nós que permanecem na fronteira ou o os nós explorados, no caso do mecanismo de procura por profundidade e procura em grafo, respetivamente.

Para terminar e executar a simulação, criou-se o módulo *teste_plan.py*. É então criado um array de *strings* das localidades e de seguida uma lista de todas as ligações. De seguida é criada uma instância do *PlaneadorTrajecto()*, e é pedido ao *user* para selecionar uma localidade de origem e uma localidade de destino. Por fim é executado o método *planear* do planeador, fornecendo a lista de ligações e as localidades desejadas pelo utilizador, depois de obter a solução é pedido que se mostre o trajeto. Podemos ver o resultado das figuras seguintes.

```
Loc-4
Loc-5
Loc-6
Loc-7
Loc-8
Loc-9
Loc-10
Escolher uma localidade de origem ->
Loc-0
Escolher uma localidade de destino ->
Loc-4
Calcular caminho de  Loc-0  a  Loc-4

Selecione o mecanismo de procura:
0-Procura Profundidade
1-Procura Profundidade Limitada
2-Procura Profundidade Iterada
3-Procura Largura
4-Procura Custo Uniforme
0
0
Calcular caminho da localidade Loc-0 a Loc-4 , com o método de  Procura Profundidade

Loc-0
Loc-7
Loc-8
Loc-4

Custo:  40
Nos processados:  3
Memorizados:  3
PS C:\00_ISEL\IASA\iasa47094_V1\iasa47094\iasa_agente>
```

Figura 13 - Resultado simulação pee – parte 1

```
Loc-0
Loc-1
Loc-2
Loc-3
Loc-4
Loc-5
Loc-6
Loc-7
Loc-8
Loc-9
Loc-10
Escolher uma localidade de origem ->
Loc-0
Escolher uma localidade de destino ->
Loc-4
Calcular caminho de  Loc-0  a  Loc-4

Selecione o mecanismo de procura:
0-Procura Profundidade
1-Procura Profundidade Limitada
2-Procura Profundidade Iterada
3-Procura Largura
4-Procura Custo Uniforme
3
3
Calcular caminho da localidade Loc-0 a Loc-4 , com o método de  Procura Largura

Loc-0
Loc-2
Loc-4

Custo:  55
Nos processados:  6
Memorizados:  9
PS C:\00_ISEL\IASA\iasa47094_V1\iasa47094\iasa_agente>
```

Figura 14 - Resultado simulação pee – parte 2

```
Selecione o mecanismo de procura:
0-Procura Profundidade
1-Procura Profundidade Limitada
2-Procura Profundidade Iterada
3-Procura Largura
4-Procura Custo Uniforme
4
4
Calcular caminho da localidade Loc-0 a Loc-4 , com o método de  Procura Custo Uniforme

Loc-0
Loc-1
Loc-3
Loc-5
Loc-9
Loc-10
Loc-4

Custo:  27
Nos processados:  10
Memorizados:  11
PS C:\00_ISEL\IASA\iasa47094_V1\iasa47094\iasa_agente>
```

Figura 15 - Resultado simulação pee – parte 3

Em todas as experiências escolheu se como localização inicial Loc-0 e final Loc-4. Na *Figura 13* utilizou se o mecanismo Procura Profundidade, na *Figura 14* o mecanismo Procura Largura e na *Figura 15* o mecanismo Procura Custo Uniforme.

3.4. Parte 9

Seguindo o tema dos Agentes Deliberativos (item 0) implementou-se um controlo deliberativo. Tal como o controlo reativo, criou-se uma classe *ControloDelib()* que estende a classe *Controlo* da *sae*. Como referido na parte teórica, este agente vai ter uma representação do modelo do mundo, por isso no construtor vai se criar uma instância da classe *ModeloMundo()*. Para alem disso, o agente vai criar um plano, assim sendo, é passado como argumento um objeto que implemente a interface *Planeador()*.

Na classe do controlo deliberativo, no método *processar()*, vai se seguir o raciocínio prático do agente deliberativo, por isso, o agente vai assimilar a perceção, atualizando o modelo do mundo, depois vai reconsiderar , ou seja , verificar se houve alteração do modelo do mundo ou se o plano ainda é válido, se o modelo sofreu alterações ou o plano já não é válido, o agente delibera, sendo que neste método o agente atualiza o conjunto de objetivos , sendo que o estado entra na lista de este for um alvo, e se houver objetivos , ordena-se os objetivos por ordem de menor distancia entre o estado onde o agente se encontra. Após deliberar, o agente planeia, delegando essa tarefa a função *planear()* do planeador. Por fim mostra se o plano e executa-se, onde se vai buscar o próximo passo da solução, para percorrer o percurso até ao destino final, através da do método do *planeador()*, *obter_accao()*, e é devolvido a ação a ser executada.

A classe *ModeloMundo()* implementa a interface *ModeloPlan()* que vai representar o espaço onde o agente se encontra, sendo que o espaço é constituído por elementos como obstáculos e alvos, que são guardados num dicionário de elementos cuja *key* é a posição do elemento. É guardado os operadores, estes que são da classe *OperadorMover()*, e um atributo *boolean* que indica se o modelo do mundo foi alterado ou não. Quando se executa o método de atualizar o modelo através de uma perceção, altera-se o estado e verifica se se os elementos percecionados são iguais aos elementos guardados no modelo do mundo, caso sejam diferentes, tem que se alterar, atualizar os elementos do modelo do mundo para os elementos percecionados, criando novos Estados, estados estes da classe *EstadoAgente()* com as posições da perceção. Altera-se o atributo que indica se o modelo foi alterado para *True*, para assim, o agente reconsiderar, voltar a deliberar e planear.

Os operadores usados para este agente são da classe *OperadorMover()*, este que tem guardado o modelo do mundo, e receber no argumento uma direção, que vai fornecer o angulo, e vai se criar uma ação com essa direção. Ao aplicar o operador, guarda se os valores da posição do estado, calcula se o *x* e o *y* do novo estado, a partir do 'passo', vindo da ação, que se multiplica pelo cosseno do angulo da direção, para o x, ou pelo valor negativo do seno do mesmo angulo. Ao obter a posição do novo estado, cria se uma instância de *EstadoAgente()* com essa posição, e se este estiver na lista de estados do modelo do mundo retorna este estado. Para calcular o custo do operador, será o maior valor entre 1 e a distância entre o estado atual e o novo estado.

O *EstadoAgente()*, referido acima, é o estado correspondente a posição do agente. Para criar uma instância deste estado, é necessário fornecer uma posição (*tuplo* com o x e y), e o id do estado será o *hash* da posição.

3.5. Parte 10

No item 2.9, definiu-se planeamento automático como um raciocínio automático a fim de planear uma sequência de ações para um agente inteligente executar. Assim, para complementar a implementação da arquitetura de agentes deliberativos, criou-se planeadores, um planeador com base em procura em espaço de estados, *plan_pee*, e o um planeador com base em processos de *Markov*, *plan_pdm*.

Sendo que se vai implementar dois planeadores, criou se a interface *Planeador()*, onde tem o método que cria o problema, a heurística (caso seja necessário) e através desses dados, aplica a função resolver do método de procura, o que irá gerar uma solução. As classes que implementarem esta interface, têm ainda que implementar o método *obter_accao()*, método que vai executar a solução, ou seja, vai avançar na lista do percurso até chegar ao objetivo final, utilizando o método *remover_passo()* da classe *Solucao*, para que essa etapa do percurso sair da lista após ser executada. A interface tem ainda mais dois métodos abstratos, *plano_valido()*, que verifica se o plano é válido, se existe um plano, uma solução, e se o estado onde o agente se encontra é o mesmo que o estado do próximo passo da solução, e o método *terminar_plano()*, para terminar o plano, colocando a solução a *None*.

3.5.1. Planeador com base em Procura em Espaço de Estados

Para este planeador, criou-se a classe *PlanPEE()*, que implementa a interface *Planeador()*, sendo o mecanismo de procura usado é de procura melhor primeiro, que é inicializado no construtor, caso seja um mecanismo de procura informada, é necessário inicializar a heurística. Para além dos métodos que se tinham que implementar pela classe implementar a interface planeador, acrescentou-se o método *mostrar()*, onde vai criar um dicionário com os estados explorados como *key*, e o *value* o seu custo. Com o objeto *vista* vindo da *sae*, é possível mostrar visualmente o custo, e a solução, sendo que o custo está do dicionário criado.

A função heurística utilizada calcula a distância entre a posição do estado final com a posição do estado final, utilizando o método do módulo *math*, *dist()*, na classe *HeurDist()*, que implementa a interface *Heuristica()*.

Por fim, é preciso desenvolver uma classe que represente o Problema, assim sendo, criou-se a classe *ProblemaPlan()*, onde para o estado inicial, vai buscar ao estado atual do *ModeloMundo()*, tal como os operadores, e o estado final é dado como argumento no construtor.

Desenvolveu-se o módulo *teste_pee.py*, que vai simular este planeador. Testou-se utilizando os mecanismos de procura, Procura Custo Uniforme, Procura A* e Procura Sôfrega. Podemos ver nas figuras seguintes os resultados da simulação, sendo que a vermelho representa o custo, quanto menor o custo mais vermelho fica.

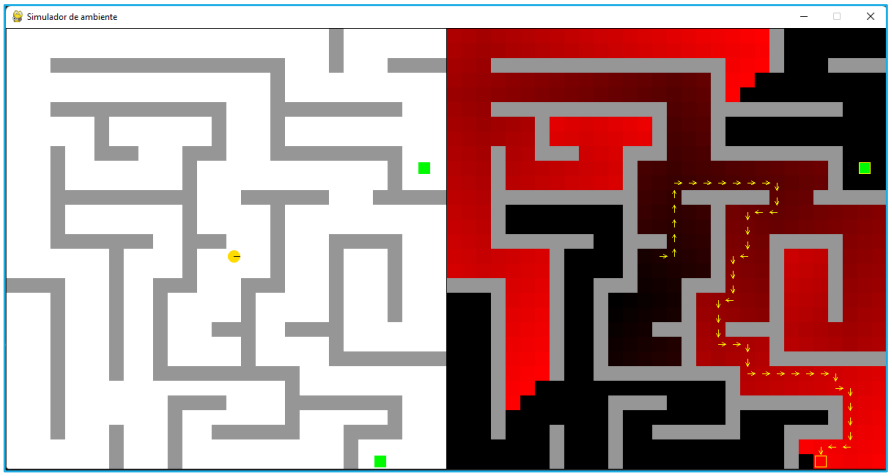

Figura 16 - Simulação com planeador pee, usando o mecanismo de procura custo uniforme

Na *Figura 16*, onde é usado o mecanismo de procura custo uniforme, podemos ver que são explorados muitos estados.

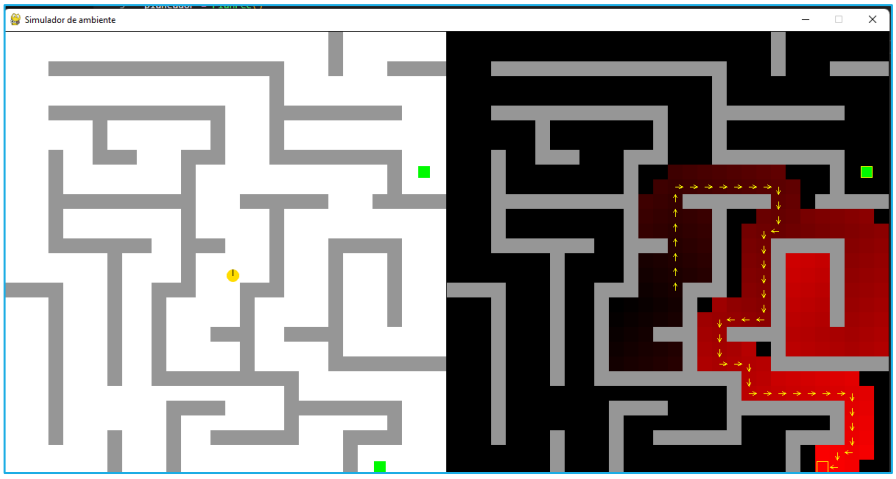

Figura 17 - Simulação com planeador pee, usando o mecanismo de procura Sôfrega

Na *Figura 17*, é usado o mecanismo de procura sôfrega, e verificamos que já não são exploramos tantos estadis como nos outros mecanismos.

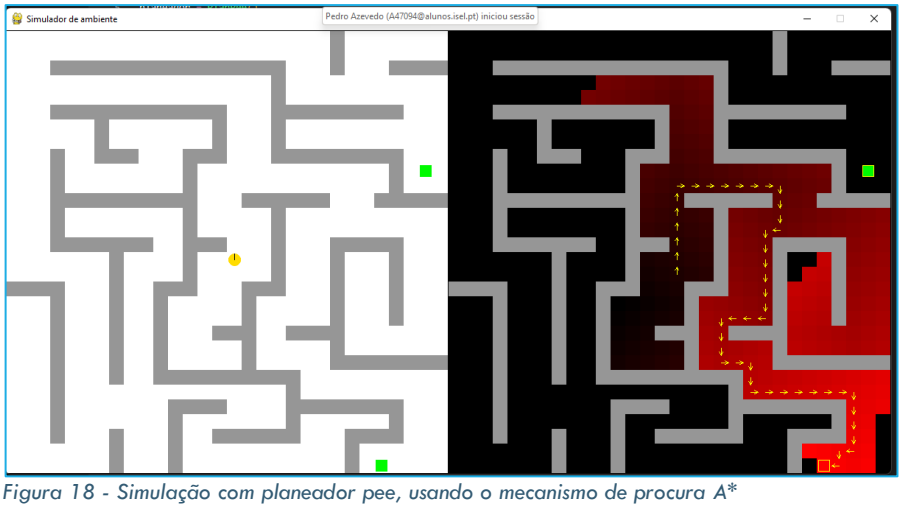

*Figura 18 - Simulação com planeador pee, usando o mecanismo de procura A**

Na *Figura 18*, foi usado o mecanimso de procura A*, onde se explora mais estados do que na procura sofrega, porém, conseguimos ver que não explora para além de estados onde têm custos baixos, ao contrário da procura custo uniforme.

3.6. Parte 11 + Parte 12

Nesta parte desenvolveu-se uma biblioteca para Processos de Decisão de *Markov* (*PDM*), com base em pontos falados no item 2.9. Para resolver problemas com estes processos, é necessário conhecer o modelo do mundo, calcular a utilidade de todas as ações, os ganhos ou perdas provenientes desse estado, e ter uma política. Assim sendo, na classe *PDM()*, implementou-se a classe *utilidade()*, onde, para cada estado, presente no modelo do mundo, vai se calcular a recompensa associada ao estado, sendo esta a recompensa máxima entre as ações possíveis do estado, usando a função presente na *Listagem 4*, onde se faz o somatório da soma da recompensa da transição do estado com a aplicação do operador, para todos os seus estados seguintes com a utilidade do estado seguinte calculada previamente multiplicado pelo fator de tempo de oportunidade multiplicando pela probabilidade de transição. Sabe-se que quanto maior o gama, o fator de tempo de oportunidade, melhor a solução, pois vai se "ver" mais o futuro, porém requer mais iterações, mais memória e mais tempo de processamento. Chega uma altura em que vemos o padrão de diferença entre a probabilidade dos estados ser sempre semelhante, podendo deduzir i estado com maior probabilidade, assim tem-se um valor delta que é calculado com através da diferença entre a utilidade anterior, com a utilidade calculada, caso o valor seja igual ou menor que o delta máximo definido previamente, significando que já não existe grande diferença entre as utilidades, devolve se o último valor de utilidade calculado.

```
'''s-Estado, a-Operador, U-Utilidade'''
def util_accao(self, s, a, U):
    '''Return double'''
    T, R, gama = self._modelo.T, self._modelo.R, self._gama
    return sum(p * (R(s, a, sn) + gama*U[sn]) for p, sn in T(s, a))
```

Listagem 4 - Cálculo da Utilidade de um estado, PDM

Para calcular a política, no método da classe *PDM()*, cria se um dicionário, onde como *key* tem se os estados, e o *value*, a ação com o valor máximo de utilidade. Quando se chama a função resolver de *PDM()*, esta calcula a utilidade e de seguida a política, utilizando a utilidade calculada, devolvendo um *tuple* com ambos os dados.

Implementou-se a interface *ModeloPDM()*, onde terá funções par ase retornar o conjunto de estados do mundo, o conjunto de ações possíveis num estado que pertence ao conjunto de estados do mundo, a função transição, da probabilidade de transição do estado dado para o estado seguinte através de um operador, e por fim a recompensa esperada na transição de i, estado para o estado seguinte através de um operador.

3.6.1. Planeador com base em Processos de Decisão de *Markov*

Como já referido no item 3.5, desenvolveu-se também um planeador com base em Processos de Decisão de *Markov*, implementando a interface *Planeador*, a classe *PlanPDM()*.

Aqui, é usado como modelo do mundo a classe *ModeloPDMPLAN()*, que implementa as interfaces *ModeloPDM()* e *ModeloPlan()*. No construtor ainda se tem como atributo um *ModeloPlan*, assim é possível reutilizar código com o fim de usar este planeador no *ControloDeliberativo()*, sem ter que alterar o mesmo, ficando compatível com o *Planeador()*. Tem que se garantir o contrato com *ModeloPDM()*, pois este modelo é dos Processos de Decisão de *Markov*. Pode-se assim, delegar os métodos *estado(), estados(),* e *operadores()*, a instância recebida de *ModeloPlan()*. Nos métodos que vêm da parte da interface *ModeloPDM()*, basta ir buscar os estados ao método delegado ao *ModeloPlan()*, assim como para ir buscar os operadores. Para o método que retorna uma lista de transições, vai se aplicar o operador ao estado, recebendo um estado seguinte, retornando uma transição com a probabilidade 1 para o estado seguinte gerado. Para a recompensa, vai se buscar o custo ao aplicar o operador ao estado para o estado seguinte, porém, caso o estado seguinte for um objetivo, é devolvido uma recompensa máxima definida previamente, mais o valor da recompensa gerado.

Já na classe *PlanPDM()*, para planear, é necessário criar um modelo PDM, onde estará o Modelo do Mundo, como referido no parágrafo anterior, assim como os objetivos. Podemos assim criar o Processo de Decisão de *Markov*, *PDM()*, executando o método *resolver()*, gerando uma utilidade e uma política. Com isto, é possível obter uma ação através da política, sendo esta um dicionário com a *key* os estados e o seu *value* o operador a ser retornado. Pode-se ver se o plano é válido, no caso de haver política e nela, o estado. Termina-se o plano colocando a política e a utilidade a *None*. Por fim, é possível visualizar a solução, utilizando o objeto *vista* da *sae*, mostrando a utilidade numa escala de verdes, sendo que maior a utilidade num estado, mais intenso fica o verde, e no caso desta ser negativa, é apresentada na escala de vermelhos. A política é representada através das setas, em cada estado.

Para a simulação do Planeamento com base em Processos de Decisão de Markov, foi criado o módulo *teste_pdm.py*, apresentando os seguintes resultados.

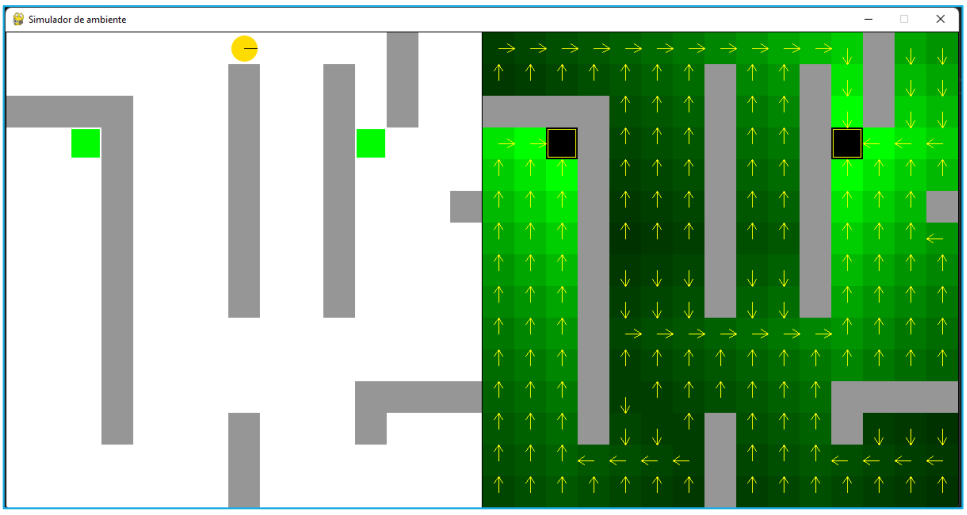

Figura 19 - Simulação do planeamento PDM Parte 1

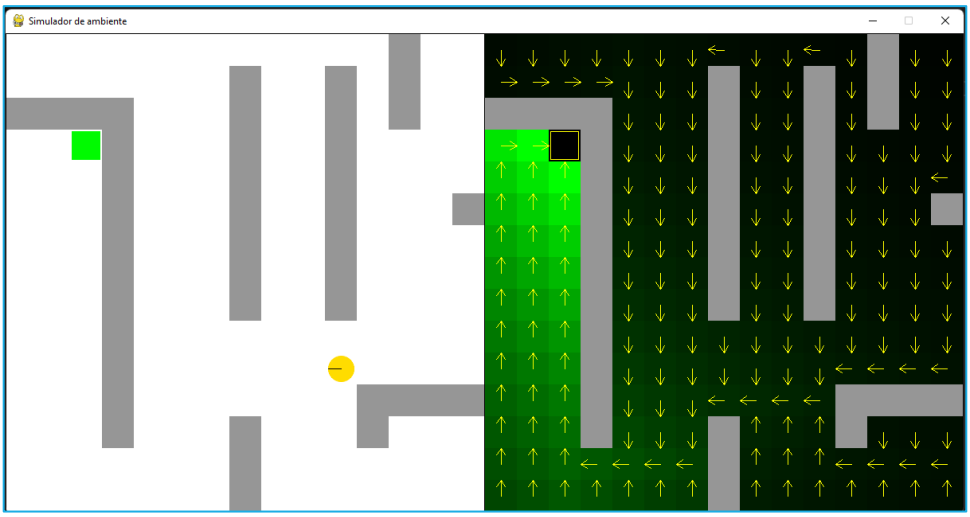

Figura 20 - Simulação do planeamento PDM Parte 2

Na *Figura 19* e *Figura 20*, podemos verificar que quando mais perto de fica do alvo temos uma utilidade maior, e que as selas descrevem o caminho certo para chegar ao alvo, indicando a direção do próximo estado que o agente deve transitar.

3.7. Parte 13

Nesta última parte do projeto, desenvolveu se um agente que tem uma aprendizagem por reforço, com base na teoria do item 2.10. Implementou uma biblioteca, onde tem a classe abstrata *AprendRef()*, e as interfaces *MemoriaAprend()* e *SelAccao()*.

Na interface *MemoriaAprend()*, as classes que a implementarem, têm que desenvolver o método que retorna o valor do Q, o valor médio para a ação e o estado recebido, implementar o método que atualize a memoria com o estado ação e o valor do Q e o método que devolva a lista de estados. Para tal, criou-se a classe *MemoriaEsparsa()*, que é um tipo de memória que vai guardar os estados e ações e associá-los ao valor Q. Sendo a memória um dicionário que associa o *tuple* do estado e ação como o calor Q, para retornar o valor Q basta utilizar o método *get()* dando o *tuple* com o estado e ação recebida. Na atualização da memória, altera-se o valor do Q no dicionário para a ação e estado dado, e adiciona-se a lista de estados o estado recebido. Esta lista pode ser pedida utilizando o método *obter_estados()*.

A interface *SelAccao()*, vai selecionar a ação a executar dependendo da aprendizagem a forma de selecionar a ação pode ser diferente, podendo haver mais do que uma forma de selecionar a ação. A classe que implementa esta interface é a *SelAccaoEGreedy()*, esta seleção é a *ε-greedy*, que dependendo do valor do ε, pode tanto explorar o mundo, como selecionar a ação que maximiza o valor estado-ação. Este valor vai ser entre 1 e 0, que define a probabilidade de executar cada ação, sendo que quanto maior o valor, mais provável vai ser o agente explorar. Na seleção da ação gera-se um número aleatório, e dependendo desse valor escolhe entre o método *accao_sefrega()*, onde vai escolher uma ação cujo valor q seja maior, podendo observar na Listagem 5. Também pode escolher o método explorar onde escolhe uma ação aleatória da lista de ações.

```
def accao_sofrega(self, s):
        '''return accao'''

        #para o caso de accoes terem o mesmo valor, a função abaixo vai
retornar sempre a primeira,
        #evitando a fazer accoes com o mesmo valor, por isso executa se uma
baralhar das accoes
        random.shuffle(self._accoes)
        #a* = argmax(a in accoes, Q(s, a))
        return max(self._accoes, key=lambda a: self._mem_aprend.q(s, a))
```

Listagem 5 - Método accao_sofrega() da classe SelAccaoEGreedy()

Na classe *AprendRef()*, vai ter um método onde o agente vai aprender, vai explorar e aprender qual o reforço do estado e da ação e atualiza o valor Q, é na base aplicar o algoritmo Q-*Learning*, tendo conhecimento de uma instância de *MemoriaAprend()* e da *SelAccao()*. Tem-se, portanto, a classe *AprendQ()*, que implementa a classe *AprendRef()*, e no método *aprender()*, vai se buscar a ação seguinte no estado seguinte, o valor de Q para o estado e a ação presente, o valor Q para o estado e ação seguintes, sendo a ação a que maximiza o Q, e por fim calcula se o valor do novo Q, atualizando na memória. Para o cálculo do valor Q para o estado e ação atual, é o valor atual do Q (estado e ação atual), somado *alfa* (valor fornecido no construtor) vezes o valor do reforço recebido da ação, mais *gama* (valor fornecido no construtor) vezes o valor do Q do próximo estado e ação, ação esta escolhida através da ação sôfrega, ou seja, a ação que maximiza o Q, este resultado subtrai-se ao valor do Q do estado e ação atuais.

Para que um agente implemente o desenvolvido previamente, criou se um controlo, *ControloAprendRef()*, e um mecanismo de aprendizagem, *MecAprend()*.

O *MecAprend()*, é uma classe fachada que vai ser usada para ocultar a complexidade dos métodos, ou seja, esta classe vai delegar às classes que agrega os métodos presentes nesta. Métodos estes como retornar os estados e o valor do Q para um estado e ação, delegados á memória, *MemoriaEsparsa()*, o método para utilizar o algoritmo Q-*Learning*, que é delegada a instância de classe *AprendQ()*, e por fim o método para selecionar a ação e o método *accao_sofrega()*, são delegados a classe *SelAccaoEGreedy()*.

A classe *ControloAprendRef()*, utiliza a aprendizagem por reforço, e desta forma , o agente não conhece o mundo, tendo este que explorar para aprender a melhor maneira de chegar ao objetivo. Para não aumentar a complexidade, esta classe usa a classe *MecAprend()*, que terá todos os métodos necessários para executar este método. É então criada uma instância da classe *MecAprend()*, dando argumento uma lista de ações com direções. No método *processar()*, vai ser executado o algoritmo Q-*Learning*, é então percecionado o próximo estado, que é do tipo da classe *EstadoAgente()*. Através da perceção é gerado um reforço, sendo este adicionado ao valor negativo do reforço mínimo (pois, para gerar um reforço, é porque houve uma ação e haverá uma recompensa negativa) com o valor da recompensa máxima, caso haja uma recolha de alvo, ou então subtraído a recompensa máxima caso não haja recolha (valores de recompensa máximo e mínimo, são definidos no construtor). Após gerado o reforço, executa-se o método *aprender()*, utilizando o algoritmo, e depois vai se gerar a ação seguinte, alterando o estado atual para o estado seguinte e a ação atual para a ação seguinte, para depois repetir o processo. Antes de retornar a ação seguinte, executa-se o método *mostrar()*, que vai representar graficamente, mostrando o valor do Q no estado da ação que maximiza a função Q, a ação sôfrega, para todos os estados, e é mostrado a politica para cada estado, sendo esta a ação seguinte da ação sôfrega.

Para fazer a simulação foi criado o módulo *teste_apremd_ref.py*, com os seguintes resultados.

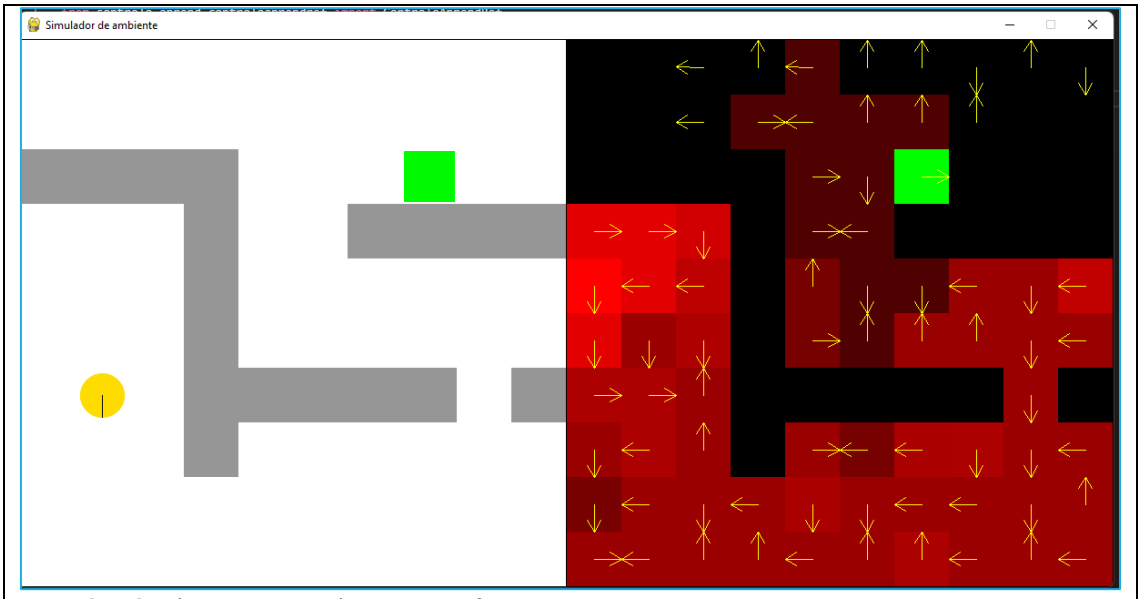

Figura 21 - Simulação com aprendizagem por reforço Parte1

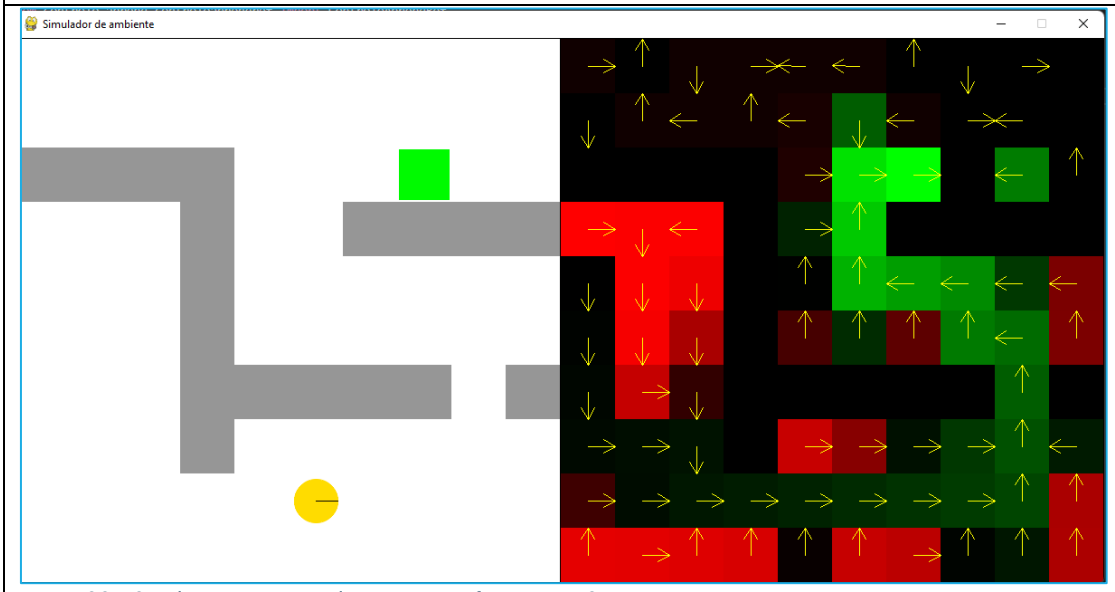

Figura 22 - Simulação com aprendizagem por reforço Parte 2

Na *Figura 21*, o agente explorou até que chegou ao alvo, nesse instante, encontrou algo onde o tivesse um valor Q positivo, e voltou ao inicio, não esquecendo o que já aprendeu, e a medida que o agente vai encontrado o alvo, mesmo que seja por caminhos diferentes, este começa a desenha a rota ideal, como vemos já na *Figura 22.*

4. Proposta de Revisão do Projeto Realizado

Nesta secção apresenta-se alguns aspetos a melhorar ao longo do projeto.

4.1. Redundância

Durante a realização do projeto, foi implementado código que, em vez de ser escrito numa só linha, era escrito em várias, como vemos o exemplo *Listagem 6* e *Listagem 7*.

```
public void executar(){
    Percecao percecao = percecionar();
    Accao accao = controlo.processar(percecao);
    atuar(accao);
}
```

Listagem 6 - Método executar, classe Personagem Parte 1 do projeto - antes

```
public void executar(){
    atuar(controlo.processar(percecionar()));
}
```

Listagem 7 - Método executar, classe Personagem Parte 1 do projeto - depois

Nas listagens apresentadas, vemos que foi possível escrever 3 linhas de código numa só. Por um lado, pode ser benéfico fazer esta alteração, pois se em todos os casos reduzir as linhas de código, no final temos menos complexidade e extensão de código. Por outro lado, criar variáveis que depois são usadas na linha exatamente a seguir e não são usadas de novo, pode ser bom, pois tem se uma melhor compreensão do código. Outro exemplo de redundância nas *Listagem 8* e *Listagem 9*, sendo que a primeira era como esta o método *activar()* da classe *Explorar()*, implementada na parte 4,e a segunda é a simplificação.

```
class Explorar(Comportamento):
    def activar(self, percepcao):
        direccao = choice(list(Direccao))
        resposta = RespostaMover(direccao)
        return resposta.activar(percepcao)
```

Listagem 8 - método activar() da classe Explorar() - antes

```
class Explorar(Comportamento):
    def activar(self, percepcao):
        return RespostaMover(choice(list(Direccao))).activar(percepcao)
```

Listagem 9 - método activar() da classe Explorar() - depois

Outro exemplo de erro de redundância ocorrido, porém poucas vezes, quando numa condição, se verifica se existe elemento, durante a implementação colocou se como esta na *Listagem 10*. Contudo, em python não é preciso ver se é diferente de *None*, basta colocar o nome da variável, que a condição retorna *True* caso a variável seja diferente de *None*.

```
if direccao_livre != None:
…
```

Listagem 10 - Condição do método activar() da classe RespostaEvitar() – antes

5. Conclusão

O projeto realizado pode ser dividido em partes, tem por base Agente Inteligentes, e estes podem ser desenvolvidos com diversas arquiteturas com diferentes comportamentos.

Primeiro, os agentes podem ter uma arquitetura de Agentes Reativos que após percecionarem, e ocorrer um evento, o agente desencadeia logo uma ação. Simulou-se esta arquitetura através de um agente que está num ambiente que contém alvos para recolher, e com esta implementação, o agente percecionava o ambiente via se tinha alvos no seu "campo de visão", se sim reagia com a ação de aproximar, caso não haja alvo verifica se existe um obstáculo para se desviar, aplicando essa ação se necessário, e por fim ainda pode explorar o ambiente. Nesta parte deu para trabalhar com um agente que pode ter várias ações que dependem dos eventos exteriores, e conseguiu-se implementar um agente que decide que comportamento ter, através de hierarquia de comportamentos.

De seguida, foi implementado um agente onde consegue procurar soluções num espaço de estados, sendo que foi estudado vários mecanismos de procura, onde uns conseguem chegar a uma solução ótima e completa, enquanto outros podem nem chegar ao objetivo, contudo cada mecanismo tem vantagens e desvantagens relacionadas com a complexidade temporal e espacial. Chegando a conclusão de que os melhores mecanismos de procura são as de procura informada, por ter mais conhecimento do problema, estimando o caminho com menor custo, são estes mecanismos a Procura Sôfrega e A*, onde por um lado a Procura Sôfrega só produz soluções sub-ótimas, contudo não tem muita complexidade espacial, não explora tantos estados, por outro lado, a procura A* encontra sempre a solução ótima, contudo temais complexidade espacial do que a Sôfrega, e pode demorar algum tempo indesejado, não podendo atuar em tempo real.

Outra Arquitetura estudada e implementada é a de agentes Deliberativos, onde o agente toma decisões com base no passado, no presente e no futuro. O agente tem um raciocínio onde observa o mundo, atualiza a representação do mundo guardada na memoria, delibera analisando o futuro, criando um plano, reconsiderando caso o ambiente mude, e por fim executa o plano. Com este agente é possível encontrar a solução ótima, e flexível ao longo da execução. Sendo possível criar um plano de várias formas, por um lado usando os mecanismos de procura em espaço de Estados, e por outro lado, processos de Decisão de *Markov*, que é há base em probabilidades, em saber qual o estado seguinte que vai dar uma melhor recompensa.

Por fim foi implementado um agente com aprendizagem por reforço, onde este já não conhece o mundo, não podendo ir ao futuro, só consegue por tentativa. Desenvolveu-se um agente que ora tanto explora os estados que dão mais recompensa, os mais benéficos, como também aproveita o que aprendeu.

Nesta Unidade Curricular, para além de conhecimentos acerca de Agentes Inteligentes e de opções para estes interagirem no mundo real como no mundo virtual, adquiri conhecimentos de Engenharia de *Software*, como desenvolver código com boas práticas, métricas e princípios. No final do semestre, evolui na programação em Python, sendo que no início comecei a cometer certos erros que mais tarde deixaram de ser, como por exemplo a colocação de "*self*" como argumento dos métodos na classe.

A Unidade Curricular foi bastante útil para introduzir o mundo da Inteligência Artificial, e de perceber o quão importante é aplicar bom desenvolvimento de *software*, para não haver muita complexidade no resultado final.

6. Bibliografia – Webgrafia

1. 24 novembro 2004, John McCarthy, What Is Artificial Intelligence?
   1.1. http://jmc.stanford.edu/articles/whatisai.html
2. IMB Cloud Education, https://www.ibm.com/cloud/learn/what-is-artificial-intelligence
3. Luís Morgado, Inteligência Artificial Introdução.
4. Luís Morgado, Modelos de Dinâmica,
5. B.J. Copeland, https://www.britannica.com/technology/connectionism-artificial-intelligence
6. https://askinglot.com/what-is-behaviourist-paradigm
7. Luís Morgado, Projecto -
   7.1. Parte 1
   7.2. Parte 2
   7.3. Parte 3
   7.4. Parte 4
   7.5. Parte 5
   7.6. Parte 6
   7.7. Projecto autónomo: Biblioteca PEE (Projecto - Parte 7)
   7.8. Projecto autónomo: Planeador de trajectos (Projecto - Parte 7)
   7.9. Parte 8
   7.10. Parte 9
   7.11. Parte 10
   7.12. Parte 11
   7.13. Parte 12
   7.14. Parte 13
8. Luís Morgado, Introdução à engenharia de software
9. Luís Morgado, Arquitectura de agentes reativos – Parte 1
10. Luís Morgado, Arquitectura de agentes reativos – Parte 2
11. Luís Morgado, Arquitectura de agentes reativos – Parte 3
12. Luís Morgado, Raciocínio automático
13. Luís Morgado, Procura em espaço de estados – Parte 1
14. Luís Morgado, Procura em espaço de estados – Parte 2
15. Luís Morgado, Arquitectura de agentes deliberativos
16. Luís Morgado, Planeamento automático com base em PEE
17. Luís Morgado, Processos de decisão sequencial
18. Luís Morgado, Planeamento automático com base em PDM
19. Luís Morgado, Aprendizagem por reforço