# Licenciatura de Engenharia Informática e Multimédia

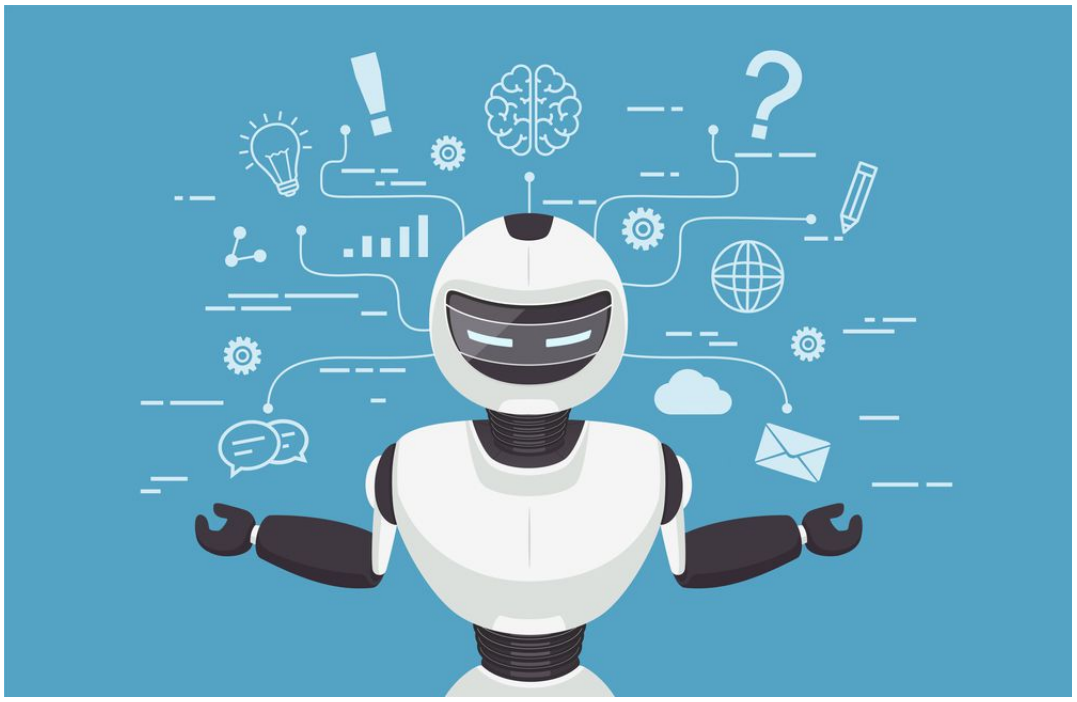
2022/2023

## Inteligência Artificial para Sistemas Autónomos

Relatório do Projeto

Docente: Luís Morgado

3/07/2023



---


Trabalho realizado por:

António Luís Ferreira, 47500
Turma 42D

# Conteúdo

# Lista de Figuras

# 1 Introdução

A Inteligência Artificial (IA) é um campo da ciência da computação que se dedica ao estudo e desenvolvimento de sistemas capazes de realizar tarefas que requerem inteligência humana. Através da aplicação de algoritmos e técnicas avançadas, a IA permite que as máquinas percebam, compreendam, raciocinem, tomem decisões e ajam de forma autónoma.

A busca por criar máquinas inteligentes remonta há décadas, com pesquisadores e cientistas explorando diferentes abordagens e conceitos. A ideia central por trás da IA é simular características humanas, como aprendizado, adaptação, solução de problemas complexos e interação com o ambiente.

A IA abrange uma ampla variedade de sub campos, como o Aprendizado de Máquina, onde as máquinas são treinadas para reconhecer padrões e tomar decisões com base em dados.

Os avanços recentes na capacidade computacional, juntamente com o aumento na disponibilidade de grandes conjuntos de dados, impulsionaram o crescimento e o impacto da IA em diversas áreas, como medicina, indústria, transporte, finanças, entretenimento e muitas outras. A IA está revolucionando a forma como interagimos com a tecnologia e está transformando pouco a pouco a sociedade de várias maneiras.

No entanto, a IA também pode trazer desafios éticos, sociais e legais. Questões relacionadas à privacidade, segurança, viés algorítmico e impacto no mercado de trabalho precisam ser abordadas e regulamentadas de maneira responsável.

À medida que a pesquisa em IA continua a avançar, a expectativa é que as máquinas se tornem cada vez mais capazes de realizar tarefas complexas, aprendam de forma autónoma e exibam um nível de inteligência comparável ao humano. A IA promete trazer benefícios significativos à humanidade, mas também demanda uma abordagem cuidadosa para garantir seu uso ético e responsável.

Este relatório tem como objetivo descrever o desenvolvimento de várias bibliotecas que facilitam a criação de aplicações de agentes autónomos, implementadas ao longo do curso de Inteligência Artificial para Sistemas Autónomos (`IASA`).

O curso está dividido em diversas partes:

- Introdução à Inteligência Artificial;
- Arquitetura de Agentes Inteligentes;
- Introdução à Engenharia de Software;
- Arquitetura de Agentes Reativos;
- Raciocínio Automático;
- Procura em Espaços de Estados;
- Arquitetura de Agentes Deliberativos;
- Aprendizagem por Reforço;

Estas partes vão ser abordados inicialmente em um nível teórico, estabelecendo todos os conceitos que permitirão uma compreensão completa do desenvolvimento subsequente descrito dentro desse contexto.

# 2 Enquadramento Teórico

## 2.1 Introdução à Inteligência Artificial

A Inteligência Artificial(IA) busca tanto compreender como reproduzir a inteligência em sistemas computacionais. Por um lado, explora-se a natureza dos fenómenos observados na inteligência humana, buscando entender seus princípios e construir modelos teóricos. Por outro lado, aplica-se esse conhecimento para desenvolver sistemas que demonstrem habilidades inteligentes, como o processamento de linguagem natural, o reconhecimento de padrões e a tomada de decisões.

Para complementar a noção de IA existem três paradigmas que têm de ser referidos, esses são:

- O **paradigma simbólico**, também conhecido como baseado em conhecimento, enfatiza o uso de representações simbólicas e regras lógicas para modelar o raciocínio e o conhecimento humano. Nessa abordagem, o foco está na manipulação de símbolos e no processamento de linguagem natural. Algoritmos de busca, sistemas especialistas e lógica formal são exemplos de técnicas utilizadas nesse paradigma;
- O **paradigma conexionista**, por sua vez, é baseado em redes neuronais artificiais. Inspirado pelo funcionamento do cérebro humano, esse paradigma envolve a construção de modelos computacionais compostos por unidades de processamento interligadas, conhecidas como neurónios artificiais. Esses modelos aprendem a partir dos dados e são capazes de reconhecer padrões e realizar tarefas como classificação e reconhecimento de padrões de forma paralela e distribuída;
- O **paradigma comportamental**, é uma abordagem que se concentra no comportamento observável dos sistemas inteligentes, sem se preocupar com os processos internos ou a cognição subjacente. Ele realça a importância da análise do comportamento em resposta a estímulos e a construção de sistemas que se comportam de maneira inteligente, independentemente dos processos mentais internos.

## 2.2 Arquitetura de Agentes Inteligentes

Agentes inteligentes são entidades de software capazes de perceber o ambiente, tomar decisões e agir de forma autónoma para alcançar objetivos específicos.

Esses agentes são projetados para interagir com o ambiente de maneira inteligente, utilizando técnicas como aprendizado de máquina, raciocínio lógico e processamento de linguagem natural.

Eles podem ser encontrados em uma variedade de aplicações, desde assistentes virtuais e *chatbots* até sistemas de recomendação e carros autónomos.

A criação de agentes inteligentes envolve o desenvolvimento de algoritmos sofisticados e a integração de diversos componentes para possibilitar a perceção, o raciocínio e a tomada de decisões com base no contexto e nos objetivos do agente.

Como está exemplificado na imagem a abaixo o sistema tem de conter sensores (para obter perceções), atuadores (realizar ações) e mais importante o controlo que funciona como o cérebro do agente onde acordo com as perceções vai determinar as melhores ações possíveis a serem aplicadas.

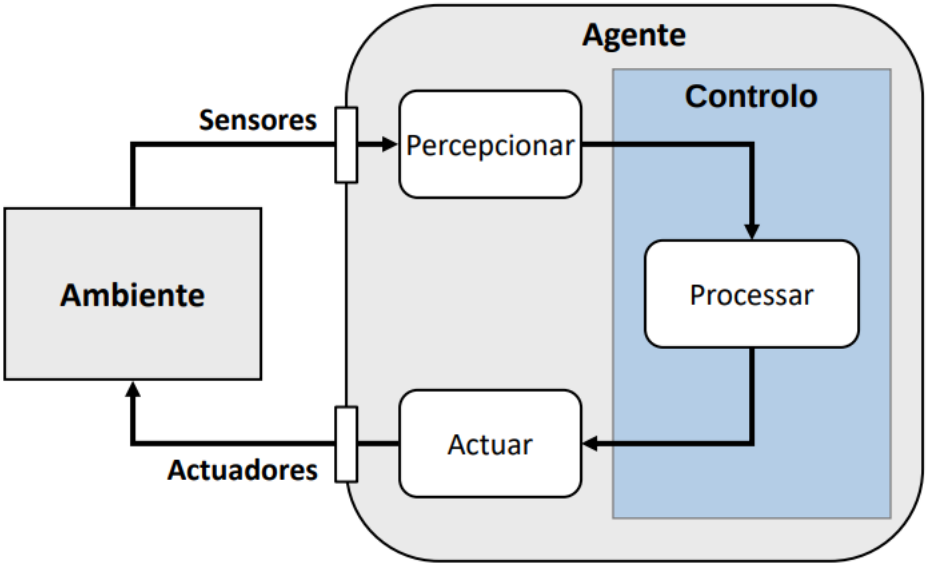



Figura 1: Estrutura de uma agente inteligente

O dependendo da implementação e funcionalidade o agente pode ter diferentes tipos de modelos de arquitetura. Sendo eles:

- Modelo **Reativo**, é representado pelo paradigma **comportamental** onde de forma reativa o agente gera uma ações com base em associações entre perceções, possível observar na figura 2;


Figura 2: Diagrama do Modelo Reativo

- Modelo **Deliberativo**, é representado pelo paradigma **simbólico**, onde é gerado com base em mecanismos de deliberação(raciocínio e tomada de decisão), com recurso a representações internas que são constituídas por representações explicitas de objetos, possível observar na figura 3;


Figura 3: Diagrama do Modelo Deliberativo

- Modelo **Híbrido**, é representado pelo comportamento gerado com base de processamentos que consistem tanto de vertentes reativas como deliberativas, possível observar na figura 4.


Figura 4: Diagrama do Modelo Híbrido

## 2.3 Introdução à Engenharia de Software

A Engenharia de Software é um conceito que se concentra no processo de análise das necessidades dos usuários e no estudo detalhado do design, desenvolvimento e teste de aplicações de software que atendam a essas necessidades. Ela aplica princípios de engenharia ao desenvolvimento de software, com o objetivo de criar soluções eficientes e seguras para um bom funcionamento.

Existem vários desafios para tal objetivo, sendo um dos desafios na engenharia de software a complexidade que os sistemas podem vir a obter. O desenvolvimento de sistemas de grande escala requer métodos diferentes dos utilizados em sistemas menores. Os métodos que podem ser aplicados em projetos menores não são escaláveis para projetos maiores, pois resultariam em sistemas confusos e desorganizados. No entanto, esse problema de complexidade pode ser resolvido por meio da aplicação de uma complexidade organizada e previamente planeada. Isso envolve o uso de padrões que fazem interligações a diferentes partes do sistema, garantindo que todas as partes tenham um propósito e funcionem em conjunto.

Com base neste principio, a arquitetura de sistemas desempenha um papel significativo. Ela deve empregar métricas, princípios e sistemas que facilitem a execução do projeto. Uma boa implementação da arquitetura de sistemas é aquela que possui baixo acoplamento, alta coesão, simplicidade e adaptabilidade. Isso pode ser alcançado por meio de técnicas como decomposição, que elimina redundâncias e sistematiza o sistema, e encapsulamento, que isola detalhes internos do sistema e reduz dependências.

Durante o desenvolvimento do projeto da Unidade Curricular, foi comum utilização modelos e esquemas para planear antecipadamente o projeto, fornecidos pelo docente. Um modelo é uma representação abstrata do sistema, que destaca os pontos essenciais e facilita a compreensão. Além disso, proporciona capacidade de previsão futura. A utilização de sistemas como o `UML` (*Unified Modeling Language*) é uma maneira eficiente de lidar com a crescente complexidade do projeto.

## 2.4 Arquitetura de Agentes Reativos

A arquitetura de agentes reativos é um paradigma de projeto utilizado na construção de agentes inteligentes. Esta abordagem realça a capacidade de resposta imediata dos agente às mudanças ao meio ambiente, sem necessariamente manter um estado interno complexo ou um planeamento robusto.

A arquitetura é especialmente adequada para ambientes dinâmicos, nos quais as condições podem mudar abruptamente. Ela permite que o agente tome decisões com base nas informações disponíveis naquele instante de tempo, sem a necessidade de processamento complexo ou considerações a longo prazo.

Os agentes também podem ser implementados com **memória**, o que ajuda a evitar outro problema comum, que é quando um agente entra em um ciclo infinito de seleção de ações sem progresso. A implementação de memória permite evitar repetir estados já explorados, garantindo que o agente não fique preso em *loops*.

Além disso, a introdução de comportamentos que ”evitam o passado”, como foi abordado em aula, permite ao agente representar e evitar situações conhecidas, gerando forças virtuais repulsivas para áreas já exploradas.

Com a introdução da memória, as reações do agente não se baseiam apenas nas perceções atuais, mas também na preservação de um estado interno. É necessário implementar regras para alterações de estado, o que envolve a criação de uma máquina de estados interna.

Essa máquina de estados vai permitir a evolução do estado ao longo do tempo, levando em consideração memórias de perceções passadas. Isso possibilita comportamentos mais complexos e com continuidade temporal. No entanto, essa abordagem aumenta a complexidade espacial e computacional, embora não permita representações extremamente complexas, possível observar na figura 5.

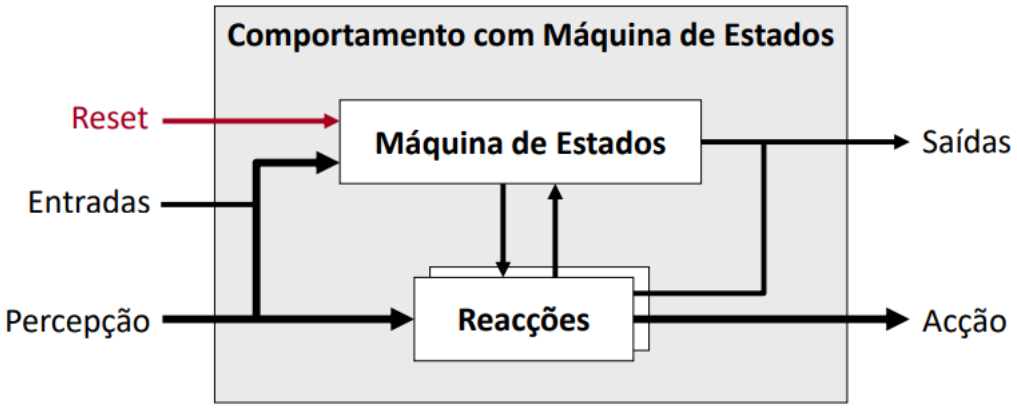



Figura 5: Máquina de Estados Interna

## 2.5 Raciocínio Automático

O raciocínio automático consiste no desenvolvimento de algoritmos e sistemas capazes de tomar decisões e inferências de forma automática, sem intervenção humana direta. Tem como objetivo habilitar máquinas a realizar processos de raciocínio semelhantes aos realizados pelos humanos, permitindo que elas analisem informações, deduzam conclusões e tomem decisões lógicas com base em regras e conhecimentos prévios.

Os sistemas de raciocínio automático são projetados para processar grandes quantidades de dados e informações, aplicando técnicas como lógica, inferência probabilística, aprendizado de máquina e representação de conhecimento. Eles são capazes de seguir uma sequência de etapas lógicas para chegar a uma conclusão ou solução para um determinado problema.

De acordo com o conhecimento do âmbito do problema o processo computacional vai fabricar possíveis conclusões baseadas no tema. A **inferência** é processo de manipulação da representação do conhecimento para obter conclusões.

Para o sistema realizar **inferências** é necessário enunciar e caracterizar algumas **noções envolvidas**, sendo elas:

- **Estado**,
  - Representa uma configuração de um sistema ou problema;
  - Identificação única;
  - Espaço de estados, conjunto de estados e de transições de estado (representado sob a forma de um grafo)
- **Operador**,
  - Representa ação;
  - Gera transformação de estado (operador.aplicar: estado $\rightarrow$ estado);
- **Problema**,
  - Estado inicial;
  - Operadores possíveis para o mecanismo usar;
  - Objetivos (ou função objetivo: estado $\rightarrow$ True, False);
- **Mecanismo de Raciocínio**,
  - Exploração de opções possíveis para encontrar uma solução através de simulação prospetiva, tendo por base uma representação interna do problema.

## 2.6 Procura em Espaços de Estados

O mecanismo de busca em espaço de estados envolve partir de um estado inicial e aplicar todos os operadores disponíveis para gerar novos estados, representados como nós em uma Árvore de Busca. **Um nó representa um elemento individual e tem referências para seus nós filhos. Cada nó em uma árvore de procura, exceto o nó raiz, tem um nó pai, que é o nó do qual ele é descendente direto. Os nós folha, por sua vez, são os nós que não têm filhos.**Esses nós podem ser classificados como abertos ou fechados, dependendo se foram ou não explorados, possível observar na figura 6.

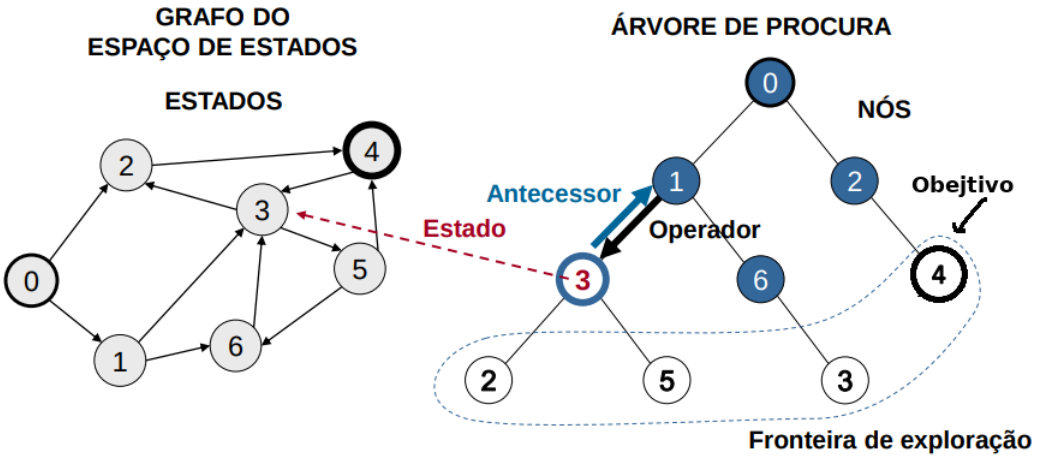



Figura 6: Grafo do Espaço de Estados e Árvore de Procura

A busca ocorre até que um estado objetivo seja encontrado. Caso isso aconteça, uma solução para o problema foi encontrada. Caso contrário, o processo é repetido até encontrar uma solução ou determinar que não há solução. A eficácia de um mecanismo de busca pode ser avaliada em termos de: tempo, espaço, completo e ótimo.

- **completo**, se o método de procura garante que a solução seja encontrada, em caso de ela existir;
- **ótimo**, se o método de procura garante que a solução seja encontrada é a melhor, em caso de haver múltiplas soluções;
- **tempo**, tempo necessário para encontrar a solução (complexidade temporal);
- **espaço**, memória necessária para encontrar uma solução (complexidade espacial);

No ponto seguinte é desenvolvido os Mecanismos de Procura que são usados no projeto.

## 2.7 Mecanismos de Procura

### 2.7.1 Procura em Profundidade

O mecanismo, tem inicio a partir de um nó inicial e explora continuamente os nós filhos mais profundos antes de retroceder. Ou seja, o algoritmo segue um caminho o mais profundo possível na estrutura antes de voltar e explorar outras opções. Utiliza uma abordagem de pilha (`LIFO`- *Last-In, First-Out*) para armazenar os nós a serem explorados. Ao encontrar um nó, ele o marca como visitado e adiciona seus nós filhos na pilha. Em seguida, continua explorando o nó mais recentemente adicionado à pilha, repetindo o processo até não haver mais nós fechados ou até encontrar o estado objetivo.

Uma característica importante do Mecanismo de Procura em Profundidade é que ele pode entrar em *loops* se não houver mecanismos para evitar a revisitação de estados já explorados. Para evitar *loops* podem ser implementados duas variantes, sendo elas:

- **Procura em Profundidade Limitada**

  O mecanismo define um limite máximo de profundidade para a busca em cada iteração, independentemente do tamanho ou da estrutura da árvore. Esse limite pode ser definido pelo usuário ou determinado pelo contexto do problema.

  Ao atingir o limite de profundidade em uma iteração, o algoritmo retrocede e explora outros caminhos que ainda não foram visitados. Esse processo continua até que a solução seja encontrada ou até que todos os caminhos tenham sido explorados dentro do limite de profundidade.

- **Procura em Profundidade Iterativa**

  Este mecanismo permite que a busca seja realizada em profundidades crescentes, começando com uma profundidade inicial mínima e vai aumentando gradualmente até atingir uma profundidade máxima, a ser definida.

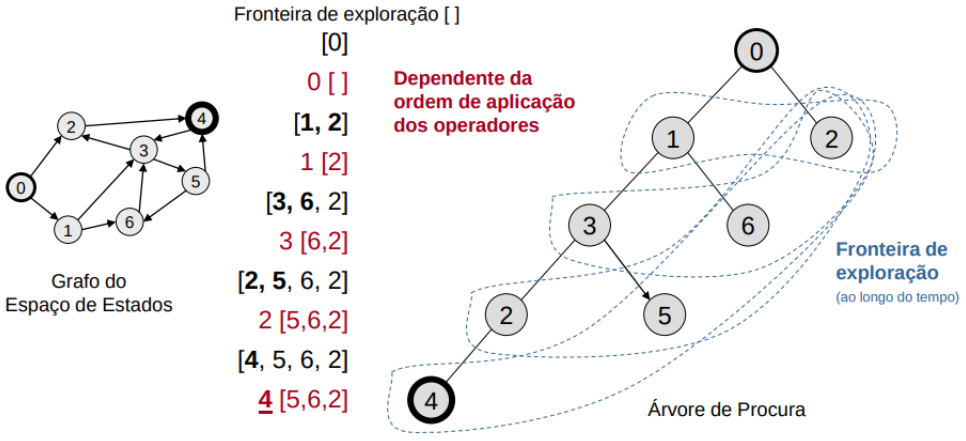



Figura 7: Grafo do Espaço de Estados e Árvore de Procura na Procura em Profundidade

### 2.7.2 Procura Em largura

Na procura em largura, a procura decorre explorando primeiro os nós mais antigos primeiros a ser gerados), levando à exploração exaustiva de cada nível de procura antes da exploração de nós a um nível de maior profundidade.

Por outro lado, a busca em largura pode exigir uma quantidade grande de memória, pois todos os nós de um determinado nível são armazenados. E em casos de grafos ou árvores muito grandes, a busca em largura pode se tornar impraticável devido à quantidade de nós a serem explorados.

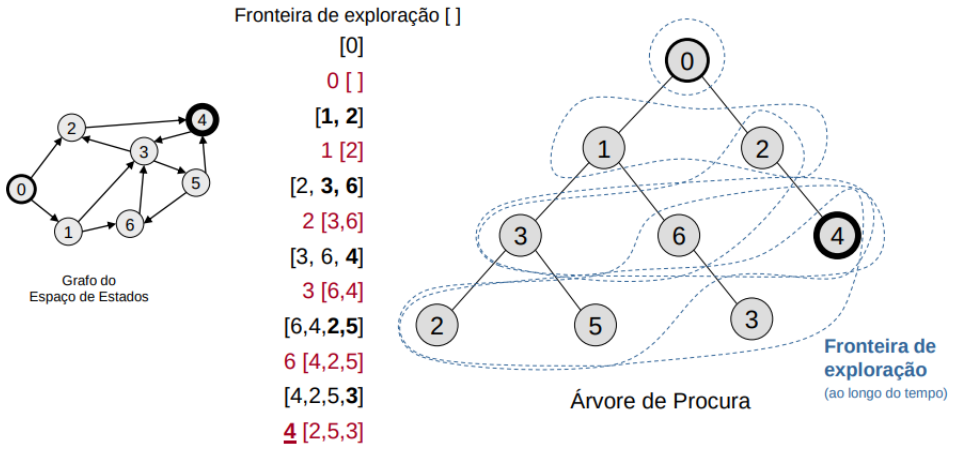



Figura 8: Grafo do Espaço de Estados e Árvore de Procura na Procura em Largura

### 2.7.3 Procura de Melhor Primeiro

A Procura Melhor Primeiro com recurso à função `f(n)`, a função retorna uma avaliação de cada nó `n`, a fronteira de exploração encontra-se organizada por prioridade (ordem crescente), quanto menor o valor de `f((n)` mais promissor é o nó.

Este mecanismo tanto pode ser **informada** como **não informada**. No caso de ser **informada** a função **heurística** com o objetivo de obter uma estimativa do custo do percurso entre dois nós com base no conhecimento do domínio do problema, ou seja neste contexto irá dizer como organizar a fronteira de exploração.

Ao contrário, na versão **não informada** o mecanismo não tira proveito do conhecimento do domínio do problema para ordenar a fronteira de exploração, recorrendo a uma procura exaustiva.

### 2.7.4 Procura de Custo Uniforme

No Mecanismo de Procura de Custo Uniforme é um método **não informado**, `f(n) = g(n)`. Consiste na implementação de uma estratégia de controlo que consiste em explorar primeiro os caminhos com menor custo. Logo, o custo que na realidade estará associado à transição até esse estado.

### 2.7.5 Procura A*

No mecanismo de Procura A* tem como objetivo encontrar o caminho menor entre os estados inicial e final, `f(n) = g(n) + h(n)`, onde `g(n)` é a soma do custo acumulado do caminho até ao nó atual e `h(n)` é uma **heurística** que estima o custo restante até o objetivo.

Uma **heurística** deve ser admissível, ou seja, nunca super estimar o custo real do caminho. Em case de ser admissível, a Procura A* garante encontrar uma solução ótima, desde que o espaço de estados seja finito.

Também utiliza uma estrutura de dados chamada fila de prioridade, que armazena os nós a serem explorados, ordenados pelo valor da função de avaliação `f(n)`. Isso permite que os nós mais promissores, com menor valor de `f(n)`, sejam explorados primeiro.

### 2.7.6 Procura Sôfrega

No mecanismo Procura Sôfrega, onde somente tem em conta a própria heurística `(h(n))`. A Heurística **não** vai ter em consideração o custo do percurso até a dado nó explorado, o único objetivo é minimizar o custo local. Obtém **soluções não ótimas**, logo, não vai encontrar sempre a melhor solução para o problema, `f(n) = h(n)`.

## 2.8 Arquitetura de Agentes Deliberativos

Numa arquitetura deliberativa, a memória desempenha um papel central na geração do comportamento do agente. Em particular, é o suporte de representação do mundo e dos mecanismos de deliberação, nomeadamente, mecanismos de raciocínio e de tomada de decisão.

No processamento interno de um agente com uma arquitetura deliberativa, para além das dimensões temporais presente e passado, é possível considerar também a dimensão futuro, a qual permite antecipar estados futuros, de modo a otimizar o comportamento do agente no presente para atingir da melhor forma estados futuros que concretizam a finalidade do sistema.

Numa arquitetura de agente, a memória suporta o processamento interno associado a diferentes dimensões temporais e tipos de comportamento:

- **Passado**, tem guardado as ações em memória;
- **Presente**, com as previsões são geradas ações às perceções;
- **Futuro**, com base no passado é feita uma previsão com os dados em memória.

Aplicando ao projecto desenvolvido em aula a ordem de execução do agente será:

- Observar meio ambiente, obtém perceções;
- Atualizar a o meio ambiente em memória;
- Pensar, com recurso à memoria, prevê opções que poderão ser implementadas;
- Prevê os fins de cada opção prevista;
- Analisar, analisa as previsões e planea;
- Executa a ação;

No entanto, a implementação eficiente de uma arquitetura de agentes deliberativos poderá ser exigente, requerendo desenvolvimento de algoritmos de raciocínio eficazes, estruturas de representação de conhecimento adequadas e mecanismos de execução eficientes.

### Processos de Decisão de Markov

O Processo de *Markov* tem como finalidade prever e controlar a sequência de interações entre um agente e meio ambiente, levando em consideração um horizonte de tempo mais longo.

Com a consideração surge um problema que requer esse tipo de solução envolve o conceito de **utilidade**, que avalia o recompensa de uma determinada ação. Essa avaliação leva em conta a incerteza inerente a todas as decisões e os efeitos cumulativos ao longo do tempo, (Recompensas aditivas e Recompensas descontadas).

Uma das características essenciais deste processo é que a previsão dos estados futuros depende apenas do estado atual, o estado seguinte é independente dos estados anteriores dados os estados atuais. Essa propriedade é conhecida como "`propriedade de Markov`"e é fundamental para a implementação do Processo de Decisão de *Markov*.

Na implementação deste processo, é necessário simular diferentes sequências de ações, estados e recompensas que podem ocorrer no futuro. Essa representação é chamada de **Cadeia de Markov**. A utilidade, por sua vez, é calculada levando em consideração o efeito cumulativo da evolução da situação, e pode estar associada a uma taxa de desconto, geralmente representada por y (um valor entre 0 e 1).

Outro conceito importante é a **política comportamental**, representa a estratégia de seleção de ações em cada estado. A política pode ser determinística, defini uma ação específica para cada estado, ou não determinística, permitindo uma distribuição de probabilidade entre as ações possíveis.

## 2.9 Aprendizagem por Reforço

A aprendizagem por reforço é uma abordagem de aprendizado de máquina em que um agente aprende a tomar decisões autónomas em um meio ambiente para maximizar uma recompensa acumulativa ao longo do tempo.

O agente interage com o meio ambiente e toma decisões com base em sua observação do estado atual do ambiente. Após cada ação, o agente recebe um *feedback* na forma de uma recompensa, que indica o quão bom foi o resultado da ação. Tem como objetivo o agente aprender uma política de ações que maximize a recompensa acumulada ao longo do tempo.

Este tipo de aprendizagem difere dos outros tipos de aprendizagem da máquina, como o supervisionado e o não supervisionado, porque não requer um conjunto prévio de dados identificados. Em vez disso, o agente explora o ambiente, aprendendo a partir das próprias interações.

Para realizar a aprendizagem por reforço, o agente utiliza algoritmos de aprendizado que buscam aprender a melhor política de ações. Esses algoritmos podem ser baseados em métodos de busca, como a Programação Dinâmica, ou em métodos baseados em aproximação, como os algoritmos de aprendizagem em profundidade.

O agente tem de explorar diferentes ações para descobrir as mais lucrativas, mas também precisa aproveitar o conhecimento adquirido para tomar decisões mais otimizadas. Além disso, a aprendizagem por reforço envolve a noção de desconto temporal, as recompensas futuras têm menos peso do que recompensas imediatas.

# 3 Projeto Realizado

Com a teoria da unidade curricular segue-se para a descrição da implementação de cada um dos projetos explicando a sua relação com o que até agora foi explicado, com a exceção da **Aprendizagem por Reforço** (não implementada).

Os subcapítulos seguintes estão organizados por ordem de implementação realizado ao longo do semestre.

## 3.1 Jogo (iasa_jogo)

A primeira **Parte do Projeto** com base num modelo de iteração é implementada em `JAVA`, tem como objetivo fazer a introdução à Engenharia de *Software* bem como inicializar a aplicação de arquitetura com base nos diagramas `UML` e a inicialização de máquinas de Estado, é possível observar a organização do projeto na figura 9.

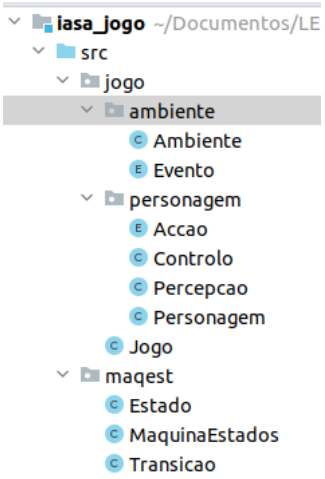



Figura 9: Estrutura do Projecto iasa_jogo

O programa ao ser executado vai pedir a introdução de um carácter por parte do utilizador, cada caráter refere a um possível evento que pode acontecer no Ambiente.

Sendo eles `SILENCIO, RUIDO, ANIMAL, FUGA, FOTOGRAFIA, TERMINAR.`

O meio ambiente, em base no seu estado atual, vai evoluir com relação ao evento, mudando para outro estado.

Seguintemente é a vez da Personagem reagir ao meio que a rodeia, com base na sua Perceção do mesmo, executando uma nova ação: mudando de estado.

## 3.2 Agente (iasa_agente)

Os subcapítulos seguintes estão implementados em PYTHON dentro do projeto iasa_agente. Também foi fornecido a biblioteca SAE (Simulação de Ambiente em Execução) para permitir a amostragem em modo gráfico da execução dos projetos.

O modo gráfico é constituido por:

- **branco**, meio em que o agente pode circular;
- **cinzento**, é um obstáculo logo o agente não pode passar por cima;
- **verde**, é um alvo;
- **amarelo**, é o agente.

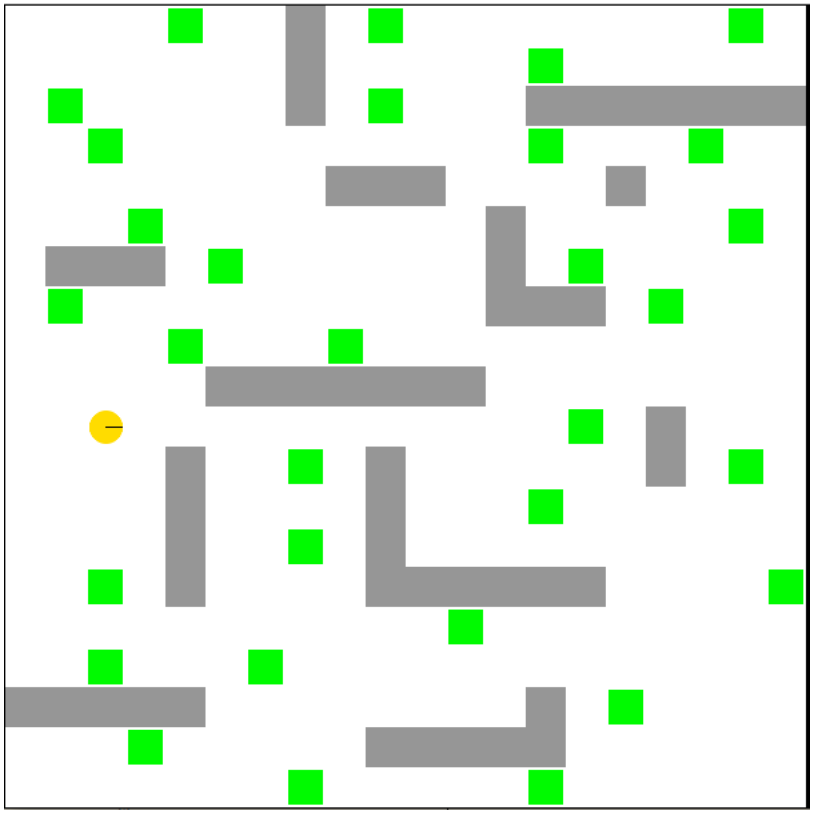


Figura 10: Demonstração do modo gráfico com os seus constituintes

## 3.3 Esquemas Comportamentais Reativos (ecr)

O funcionamento da `ECR` pode ser observado na imagem seguinte.

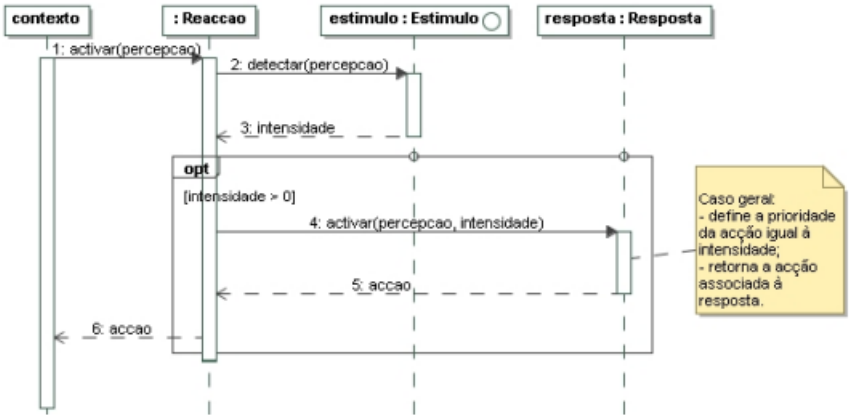



Figura 11: Diagrama UML do funcionamento da biblioteca ECR

Este modulo terminava com um executável, sendo ele o `teste_react.py`. Ao executar o ficheiro `py` vai ser criado um objeto da classe `Classe ControloReact` e usando a mesma para a execução da simulação.

A `Classe ControloReact` está responsável pelo **controlo reativo** do agente, ela herda de Comportamento e ao longo da sua execução chama o método `processar()` que de acordo com a perceção recebida vai ativar o comportamento.

Sabe-se que o agente apenas é inicializado com uma perceção do meio ambiente. Ao ativar uma reação, vai detetar a **intensidade** do estimulo do meio que o rodeia. Em caso dessa intensidade for maior que 0, vai ser ativado uma resposta. Consequentemente essa ativação vai gerar uma ação.

De forma a estruturar o código sabendo que pode haver vários tipos de comportamento foi criado a interface `Comportamento` de modo a diminuir complexidade. Tem definido um método `activar()` que recebe uma perceção, com o intuito de produzir ações. Esta interface é usada pelo os comportamentos diretos/simples (`Classe Reaccao`) e pelos comportamentos Compostos (`Classe ComportComp`).

No caso dos comportamentos compostos tem de ser implementada uma hierarquia (`Classe Hierarquia`) e uma prioridade `Classe Prioridade` que possuem a definição do método abstrato `selecionar_accao(Accao accao)` definido na `Classe Comport Comp`.

Já foi dito que o tem perceções que de acordo com os comportamentos gerão respostas, mas primeiro ainda é necessário identificar quais são as reações que o agente deve ter. Foi fornecido um diagrama `UML` que tem a ordem de implementação da `Classe Recolher` (a classe herda de hierarquia), figura abaixo.

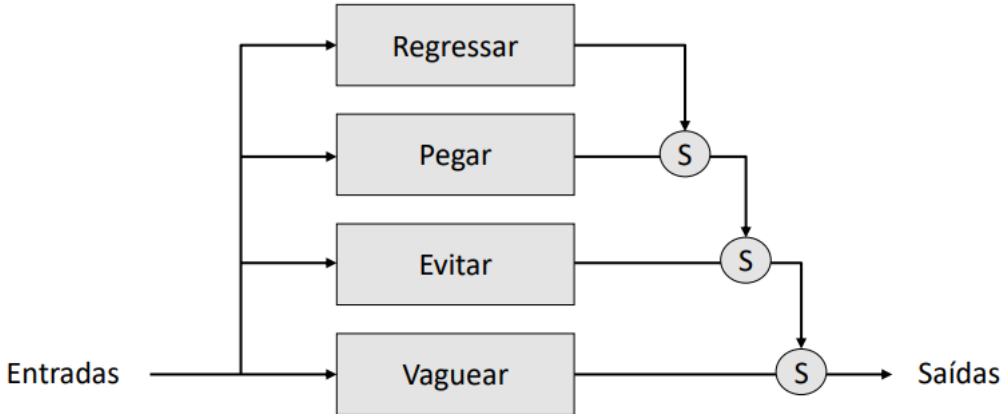



Figura 12: Diagrama UML com Tarefa de prospeção

A `Classe Recolher` vai ter os três comportamentos primários do agente, sendo eles:

- `AproximarAlvo();`
- `EvitarObst();`
- `Explorar()`.

Como é uma **hierarquia** o mecanismo de seleção é o primeiro na ordem de comportamentos. Portanto, o agente tem como principal objetivo recolher alvos (`AproximarAlvo()`), como o ambiente tem obstáculos é imperativo que ele se consiga desviar (`EvitarObst()`), por fim, em último caso de não existirem alvos o agente deve apenas explorar o ambiente aleatoriamente (`Explorar()`).

Respeitando os requisitos (organização dos comportamentos) é possível obter o resultado desejado: uma execução rápida em que o agente coleta os alvos mais próximos, contornando os obstáculos do meio ambiente. Com tudo, há uma particularidade que seria previsível dada a distinção entre **reativo** e **deliberativo**, quando o agente comete um erro, ele não possui nenhuma forma de evitar que o mesmo erro seja repetido, muitas vezes parecendo que ele ”se perde”na busca por obstáculos. Embora essa implementação tenha uma complexidade de código e computacional relativamente baixa, o tempo necessário para coletar os alvos não é otimizado.

## 3.4 Biblioteca Modulação de Problema (mod)

Esta biblioteca define as três noções envolvidas no Raciocínio Automático (**Estado**, **Operador** e **Problema**. Como esta biblioteca vai ser usada por outras os conceitos definidos nela são interfaces para posteriormente poderem ser complementadas de acordo o problema que se esteja a resolver.

De frisar que na `Interface Estado()` é redefinido o método `__hash__()` para que devolva o inteiro identificativo do objeto que o chama, o identificador único fica guardado na variável `id_valor()` em memória.

Nesta mesma biblioteca implementou-se a `Classe EstadoAgente(Estado)`, tem com principal objetivo descrever a posição que o agente toma no meio ambiente. Guarda no seu construtor a posição (tuplo com as coordenadas 2D) e o seu identificador único é o *hash value* do tuplo das coordenadas.

Já a `Interface Operador()` representa a transição entre estados, tem dois métodos:

- `aplicar()`, irá retornar o estado gerado da sua aplicado sobre um estado;
- `custo()`, um estado e o seu estado seguinte resulta o custo da operação.

Por fim, a `Interface Problema()`, define que um problema deve ter o estado inicial e quais operadores pode vir a usar, tem um método abstrato `objectivo(Estado estado)` a ser implementado de acordo com o problema em causa, também define propriedades do tipo *read-only* (só leitura).

## 3.5 Biblioteca de Procura em Espaço de Estados (pee)

Esta biblioteca é a mais extensa do projeto onde são definidos: **nó**, **solução**, os **mecanismos de procura** (tanto as não informadas como as informadas), as **fronteiras**, os **avaliadores**.

Tanto a `Classe No()` como a `Classe Solucao()` são duas classes de constituintes da representação do problema.

A `Classe No()` define a noção de estado, operador e profundidade se for um nó raiz tanto o operador como o antecessor está a *None*.

Já a `Classe Solucao()` descreve o percurso feito desde o estado inicial até ao objetivo, constituido por uma lista de nós (`percurso`).

Neste projeto são implementadas diferentes tipos de **fronteira** tanto que é necessário criar uma `Interface Fronteira()` para motivos de simplificação, na interface é definido o construtor que inicializa uma lista de nós a vazio, duas *properties* uma que vê se a fronteira está vazia e outra retorna o tamanho da fronteira, implementa o método `remover()` (consiste ir à fronteira e remover o índice 0), por fim é instanciado o método `inserir(No no)` que de acordo com o tipo de fronteira são implementados. Neste projeto foram realizadas três tipos de fronteiras sendo elas:

- **Fronteira *first in first out* (fifo)**,

  Como é do tipo **primeiro a entrar é o primeiro a sair lugar** no método `inserir(No no)` é feito um *append* à lista de nós.

- **Fronteira *last in first out* (lifo)**,

  Como é do tipo **útlimo a entrar é o primeiro a sair lugar** no método `inserir(No no)` é feito um *insert* do nó no inicio da lista de nós.

- **Fronteira prioridade**,

  Introduz o conceito de **avaliador** à fronteira ao contrario das outras na lista da fronteira não tem somente nós mas sim um tuplo, constituinte por prioridade e nó. a prioridade e calculada através do método prioridade definido nos avaliadores passando o nó.

Descrição dos **Mecanismos de Procura**:

- **Procura Profundidade**, através da *super class* passa a fronteira *lifo* ao construtor do **Mecanismo de Procura**, para além também implementa o método `_memorizar(No no)` faz uma inserção do nó na fronteira(inserção é chamada do método `inserir(No no)` da fronteira *lifo*);
- **Procura Profundidade Limitada**, herda de **Procura Profundidade** e introduz o conceito de profundidade máxima(valor *default* 100), só se expande se a profundidade do nó for inferior à profundidade máxima;
- **Procura Profundidade Iterativa**, herda de **Procura Profundidade** recebe um limite de profundidade e um incremento da mesma. Começa com a profundidade a 0 em caso de não chegar ao limite faz uma procura se encontrar uma solução incrementa a profundidade e retorna-a;
- **Procura Grafo**, introduz o dicionário de explorados (informação sobre os estados que já foram explorados e respetivos nós);
- **Procura Largura**, herda de **Procura Grafo** através da *super class* passa a fronteira *fifo*;
- **Procura Melhor Primeiro**, herda de **Procura Grafo** através da *super class* passa a fronteira prioridade para isso tem de receber um avaliador. No método `manter(Nó no)` existe duas condições para ser mantido, em caso de não ter sido explorado, vai ser adicionado (realizado na verificação já é feita pelo manter() da superclasse) ou no caso de já ter sido explorado mas o custo for menor a um nó do mesmo estado, vai ser substituído em explorados e adicionar numa posição mais prioritária na fronteira. (Já realizado na classe Fronteira Prioridade);
- **Procura Custo Uniforme**, herda de **Procura Melhor Primeiro** e define o avaliador (**Avaliador Custo Uniforme**)

Posteriormente foi implementada as **Procuras Informadas**, passando a descrever:

- **Procura Informada**, herda **Procura Melhor Primeiro** e é implementado a heurística de forma a obter uma estimativa do custo do percurso entre dois nós.
- **Procura A***, herda **Procura Informada** e passa o **Avaliador AA** à *class super*.
- **Procura Sôfrega**, herda **Procura Informada** e passa o **Avaliador Sôfrega** à *class super*.

Em suma é neste dois avaliadores que estão as definições das heurísticas, relembrando sendo na **Procura A*** a heurística de `f(n) = g(n) + h(n)` e na **Procura Sôfrega** `f(n) = h(n)`.

## 3.6 Planeador de Trajetos(plan_traj)

Com esta parte do projeto, tem como objetivo verificar a implementação da biblioteca `PEE`. O que se pretende é criar um planeador de trajetos entre duas localidades. Para tal com recurso à biblioteca `MOD` foi definido o **estado locadidade, operador ligacao, problema plan traj**. Na figura 13 é possível constatar as ligaçoes possíveis e quais são as localidades inicial e final.

```
LOC_INICIAL = 'loc-0'
LOC_FINAL = 'loc-4'

# array de ligações existentes
ligacoes = [
    Ligacao('loc-0', 'loc-1', 5),
    Ligacao('loc-0', 'loc-2', 25),
    Ligacao('loc-1', 'loc-3', 12),
    Ligacao('loc-1', 'loc-6', 5),
    Ligacao('loc-2', 'loc-4', 30),
    Ligacao('loc-3', 'loc-2', 10),
    Ligacao('loc-3', 'loc-5', 5),
    Ligacao('loc-4', 'loc-3', 2),
    Ligacao('loc-5', 'loc-6', 8),
    Ligacao('loc-5', 'loc-4', 10),
    Ligacao('loc-6', 'loc-3', 15),
]
```

Figura 13: Localidades com os respetivos custos

Para simplificação foi feito uma *dataclass* de nome Ligação que tinha a estrutura das ligações(origem,destino e custo). O planeador trajeto consistia numa classe com a função de instanciar o problema passando as ligações localidade inicial e final, definir o mecanismo de procura e retornar a solução.

Também foi proposto implementar a `Classe Trajecto`, basicamente consistia em tratar os dados para posteriormente mostrá-los na consola.

Como requisito, necessário testar para todos os tipos de procura não informadas e mostrar: a solução, dimensão, custo, complexidade espacial e por fim complexidade temporal. Possível observar na figura 14.

```
Mecanismo de Procura  ProcuraCustoUnif
Solução:  ['loc-0', 'loc-1', 'loc-3', 'loc-5', 'loc-4']
Dimensao:  5
Custo:  32.0
Complexidades Espacial  6
Complexidades Temporal  6

Mecanismo de Procura  ProcuraProfIter
Solução:  ['loc-0', 'loc-2', 'loc-4']
Dimensao:  3
Custo:  55.0
Complexidades Espacial  2
Complexidades Temporal  6

Mecanismo de Procura  ProcuraProfLim
Solução:  ['loc-0', 'loc-2', 'loc-4']
Dimensao:  3
Custo:  55.0
Complexidades Espacial  2
Complexidades Temporal  2

Mecanismo de Procura  ProcurarProfundidade
Solução:  ['loc-0', 'loc-2', 'loc-4']
Dimensao:  3
Custo:  55.0
Complexidades Espacial  2
Complexidades Temporal  2

Mecanismo de Procura  ProcuraLargura
Solução:  ['loc-0', 'loc-2', 'loc-4']
Dimensao:  3
Custo:  55.0
Complexidades Espacial  6
Complexidades Temporal  5
```

Figura 14: Execução do plan_traj

## 3.7 Controlo Deliberativo (control_delib)

A `classe ControloDelib` é possuidora de um dos métodos mais importantes: `processar (Percepcao percepcao)`, devolve a ação a desempenhar com base na perceção que o agente tem do mundo. É conseguindo respeitando o diagrama fornecido pelo docente, o diagrama descreve:

- Ativa a assimilação da perceção;
- Faz uma verificação se é preciso reconsiderar;
- Em caso afirmativo, ativa deliberar e planear;
- Chama o executar, retorna uma ação obtida por essa chamada.

Os restantes métodos desta classe visão em simular o modelo do mundo para se permitir fazer uma simulação do futuro com base na aplicação os operadores do domínio do mundo, reconhecendo os possíveis soluções.

Para o agente se conseguir movimentar no ambiente é importante frisar de como é implementado o aplicar do **Operador Mover**. Cada estado é uma posição, e o operador representa um movimento em dado ângulo, a posição que o novo estado (x', y') representa obtém-se aplicando o movimento pelo vetor (dx, dy) na posição que o estado atual (x, y) representa. Possível observar na imagem abaixo.

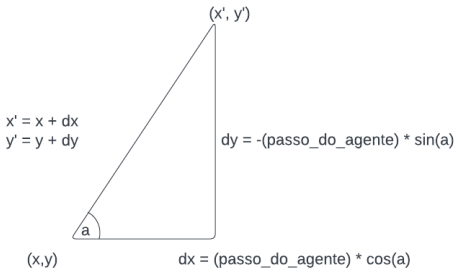



Figura 15: Esquema do aplicar do Operador Mover

## 3.8 Planeamento Automático (plan)

Com a implementação do `PlaneadorPEE` é possível fazer usar heurísticas. É importante frisar que quando o planeador executa o método `planear()` retorna um objeto a classe Plano passando a solução calculada como parâmetro.

Criou-se duas heuristicas, uma que implementa a distância de Euclidiana e outra a de *Manhattan*. A imagem seguinte mostra a execução usando o mecanismo de procura **Procura A*** três alvos no meio ambiente com as duas heurísticas.

```
Heuristica: Distancia de Manhattan
Complexidade Temporal:  202
Complexidade Espacial:  226

Heuristica: Distancia de Manhattan
Complexidade Temporal:  420
Complexidade Espacial:  233

Heuristica: Distancia de Manhattan
Complexidade Temporal:  807
Complexidade Espacial:  426

Heuristica: Distancia de Euclideana
Complexidade Temporal:  230
Complexidade Espacial:  247

Heuristica: Distancia de Euclideana
Complexidade Temporal:  463
Complexidade Espacial:  247

Heuristica: Distancia de Euclideana
Complexidade Temporal:  882
Complexidade Espacial:  438
```

Figura 16: Execução do pee

**Interpretação**:

A heurística de distância de *Manhattan* apresenta uma complexidade temporal menor em comparação com a heurística de distância euclidiana. Isso indica que a heurística de *Manhattan* encontra soluções mais rapidamente para os problemas da busca.

Já em relação à complexidade espacial, a heurística de distância de *Manhattan* também tende a ter valores menores do que a heurística euclidiana. Logo, sugere que a heurística de *Manhattan* exige menos espaço de armazenamento durante o processo de busca.

Por fim, pode-se admitir que a heurística da distância de *Manhattan* é mais eficiente tanto em termos de tempo quanto de espaço em comparação com a heurística de distância euclidiana no contexto do algoritmo de **Procura A***.

## 3.9 Biblioteca Processo de Markov(pdm)

Na biblioteca `PDM` é implementado os métodos referentes aos conceitos mais importantes já referidos, utilidade e a política.

- **Utilidade**

  Método que calcula a utilidade de todos os estados, efeito cumulativo da evolução da situação. Relembra-se que, neste modelo, tem-se acesso ao modelo e ao gama. Respeitando o algoritmo: Inicia-se a utilidade, um dicionário estado: utilidade, a zeros;

  - Vai se iterar o dicionário obtendo a maximização da utilidade da ação, correndo as ações que a este podem ser aplicadas;
  - A iteração é repetida até que a fórmula resulte na forma ótima, condição reconhecida quando já não há mais alteração face aos resultados de utilidade, este critério é auxiliado pela variável delta (variação entre duas iterações) , enquanto o delta obtido for menor que delta máximo;

- **Util_accao**

  É um método auxiliar com o final de calcular a utilidade de um único estado, vai ser útil na tomada de decisões ótimas. As utilidades dos estados podem ser determinados em função das utilidades dos estados sucessores;

- **Política**

  É um método que vai retornar uma estratégia de seleção de ação, com base numa função de utilidade, no caso não determinístico.

## 3.10 Planeamento Automático com Recurso ao Processo de Markov (plan_pdm)

Para testar a biblioteca `PDM` tal como no `PEE` foi implementado o `PlaneadorPDM` onde é definido o delta máximo e o gama a serem usados na execução.

Nas imagens abaixo mostra a importância do acerto destes imper-parâmetros para o bom funcionamento do projeto. Pode-se também constatar que quanto mais perto o agente se situa do alvo, mais forte é o tom de verde se encontra o estado(utilidade maior). As setas amarelas representam a política indicando a direção do próximo estado para o qual o agente deve transitar, quando num posicionado.

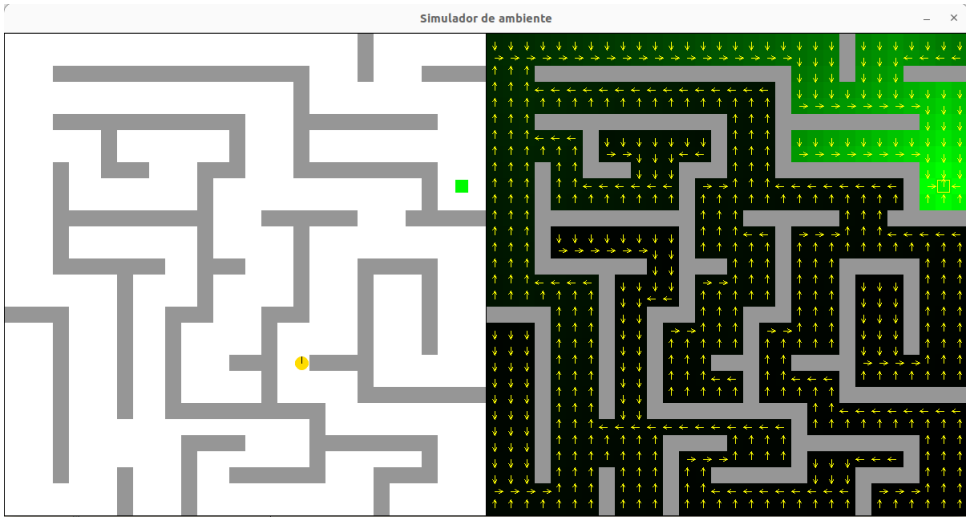



Figura 17: Execução do pdm com o gama a 0.95

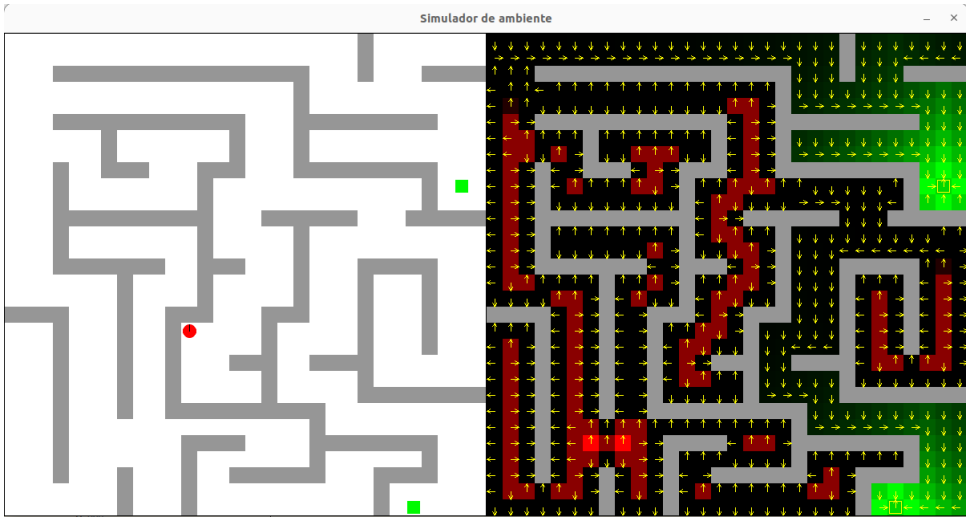



Figura 18: Execução do pdm com o gama default (0.85)

## 3.11 Exercícios Propostos pelo Docente

Para bom entendimento das bibliotecas implementadas por duas ocasiões foi proposto exercícios.

**a)** O primeiro consistia no problema de **Encher um deposito** onde o agente só poderia encher ou vazar 2/3 litros tendo em atenção de a solução não ter soluções com volumes impossíveis (exemplo: volume inicial 0 e final de 1 e a solução começar por vazar 3 L).

De acordo com a biblioteca `MOD` definiu-se os operadores ( sendo eles operador transferir continha a parte comum entre o operador vazar e encher), estado (indica o volume que o recipiente tem) o problema sabe qual o volume inicial e final e operadores possíveis (Encher(2),Encher(3),Vazar(2),Vazar(3)).

O importante a realçar deste exercício é como os operadores foram implementados:

- **Operador Transferir**, herda de `Operador()`
  - Construtor, recebe o volume atual e guarda em memória.
  - Aplicar, a ser implementado com as expessificações do encher e vazar.
  - Custo, o calculo do custo é igual para as duas consiste em ser o quadrado do modulo da subtração do volume do estado sucessor com o do atual,

    ```
    abs(estado_suc.volume - estado.volume) ** 2.
    ```

- **Operador Encher**, herda de `OperadorTransferir()`
  - Aplicar, recebe um estado e soma o volume desse estado ao volume atual(definido no operador transferir), retornando um novo estado com o resultado da soma(novo volume).
  - Redefinição do método __repr__para quando se faz *print* do volume mostrar na forma de `Encher(2)/Encher(3)`.
- **Operador Vazar**, herda de `OperadorTransferir()`
  - Aplicar, recebe um estado e subtrai o volume desse estado ao volume atual(definido no operador transferir), em caso ser menor que 0 o novo volume é 0, retornando um novo estado com o resultado da soma(novo volume).
  - Redefinição do método __repr__para quando se faz *print* do volume mostrar na forma de `Vazar(2)/vazar(3)`.

Como é possível observar na imagem 19 foi requerido aplicar diversos mecanismos de procura para isso no executável(`teste_deposito.py`) fazer vários tipos de procura e armazenálos numa lista de soluções, para posteriormente com o auxilio da nova `classe Trajecto()` (Trata a informação para mostrar os *prints* pela ordem correta).

```
    => Volume inicial 0 <=
    => Volume final 9  <=

 ProcuraCustoUnif
Solução:  [Encher(2), Encher(2), Encher(2), Encher(3)]
Dimensao:  5
Custo:  21.0

 ProcuraProfIter
Solução:  [Encher(3), Encher(3), Encher(3)]
Dimensao:  4
Custo:  27.0

 ProcuraProfLim
Solução:  [Encher(3), Encher(3), Encher(3)]
Dimensao:  4
Custo:  27.0

 ProcuraLargura
Solução:  [Encher(3), Encher(3), Encher(3)]
Dimensao:  4
Custo:  27.0
```

Figura 19: Execução do deposito

**b)**Já o segundo exercido consistia **Ordenar blocos por ordem crescente** para comparar resultados entre procuras informadas e não informadas, os mecanismos de procura usados foram o de **custo uniforme** e o **A***.

Estados blocos, como estamos a manipular lista de listas, no construtor é necessário: fazer a manipulação necessária para conseguir calcular o identificador único

`(self.__id_valor = hash(tuple( tuple(pilha) for pilha in self.__pilhas))).`

É importante frisar devido ao problema de ser uma lista com listas para os operadores conseguirem fazer as alterações necessárias e guarda-las criou-se no Estado Blocos uma propriedade só de leitura que faz *deepcopy* das pilhas.

Tal como no exercício anterior os operadores foram organizados em três partes:

- **Operador Transferir**, herda de `Operador()`
  - Construtor, recebe o numero da pilha e guarda em memória.
  - Aplicar, a ser implementado com as expessificações do empilhar e desempenhar.
  - Custo, o custo é número da pilha em guardado no construtor
- **Operador Empilhar**, herda de `OperadorTransferir()`
  - Aplicar, recebe um estado, o estado contem as pilhas. Vai às pilhas remover o número que quer empilhar no inicio da primeira pilha(remoção *pop* e inserção *insert*). Retorna um novo estado com a nova pilha alterada.
  - Redefinição do método __repr__para quando se faz *print* do volume mostrar na forma de `Empilhar(1)/Empilhar(2).`
- **Operador Desempilhar**, herda de `OperadorTransferir()`
  - Aplicar, recebe um estado, o estado contem as pilhas. Vai à primeira pilha remover o primeiro número, para de seguida inserilo na pilha desejada com recurso ao numero pilha guardado na `Class OperadorTransferir()` (remoção *pop* e inserção *insert*). Retorna um novo estado com a nova pilha alterada.
  - Redefinição do método __repr__para quando se faz *print* do volume mostrar na forma de `Empilhar(1)/Empilhar(2).`

O **Problema** recebe a configuração inicial e final, no seu construtor define os operadores possíveis

`(OperadorEmpelhar(1 ou 2),OperadorDesempelhar(1 ou 2))`,

guarda a configuração final em memória e chama o construtor da *super class* passado o estado com a configuração final e os operadores.

O objetivo é alcançado quando a primeira pilha é igual à configuração final.

A **Heurística** a ser usada pela **Procura A*** tem o objectivo de calcular a diferença entre o estado atual com objetivo.

O **Planeador** recebe a configuração inicial, final , mecanismo de procura a ser usado e a heurística(em caso de não haver o valor *default* é *None*), define o problema com as configurações recebidas e calcula a solução.

```
Mecanismo de Procura  ProcuraCustoUnif
Solucao:
[[2, 3, 1], [], []]
[[3, 1], [2], []]   Desempelhar(1)
[[1], [3, 2], []]   Desempelhar(1)
[[], [3, 2], [1]]   Desempelhar(2)
[[3], [2], [1]]   Empelhar(1)
[[2, 3], [], [1]]   Empelhar(1)
[[1, 2, 3], [], []]   Empelhar(2)

Dimensão:  7
Custo:  8.0
Complexidades Espacial  49
Complexidades Temporal  35

Mecanismo de Procura  ProcuraAA
Solucao:
[[2, 3, 1], [], []]
[[3, 1], [2], []]   Desempelhar(1)
[[1], [3, 2], []]   Desempelhar(1)
[[], [3, 2], [1]]   Desempelhar(2)
[[3], [2], [1]]   Empelhar(1)
[[2, 3], [], [1]]   Empelhar(1)
[[1, 2, 3], [], []]   Empelhar(2)

Dimensão:  7
Custo:  8.0
Complexidades Espacial  41
Complexidades Temporal  26
```

Figura 20: Execução do blocos

Ao observar a imagem 20 pode-se concluir que o motivo para as duas procuras resultarem soluções iguais é:

- O grafo não tem ciclos negativos. Tanto a procura por custo uniforme quanto a **procura A*** consideram que o grafo não contém ciclos negativos. Caso contrário, esses algoritmos podem entrar em *loops* infinitos, pois não há uma solução ótima em um grafo com ciclos negativos. Portanto, se o grafo for livre de ciclos negativos, tanto a procura por custo uniforme quanto a **procura A*** podem fornecer os mesmos resultados, encontrando o caminho mais curto.
- É de relembrar que **procura A**** é uma melhoria em relação à outra procura, pois tem a noção de heurística. A heurística fornece uma estimativa do custo restante para atingir o objetivo a partir de qualquer nó, permitindo que a **procura A*** seja mais eficiente em termos de tempo de execução em comparação com a procura por custo uniforme. Portanto, a **procura A*** é geralmente preferida quando a heurística de estimativa é aplicável e pode fornecer resultados mais rápidos e eficientes

# 4 Revisão do projeto realizado

Tendo em conta que este capítulo serve de introspeção do trabalho realizado ao longo do semestre. No respeito às entregas o código proposto em aula foi sempre implementado e comentado com a exceção de poucas ocorrências em que não deu tempo, sendo comentado a respetiva parte e entregue na semana seguinte atualizado.

No inicio também em certas alturas confirmei a existência de algo com a designação `if(xpto != None)` (xpto apenas para mera referência) podendo apenas fazer `if(xpto)` para obter o comportamento esperado de ver se existe.

Em certas partes do trabalho criou-se variáveis explicativas para o aluno entender melhor o conhecimento dos conteúdos lecionados, não influenciam em nada para a máquina fazer o suposto.

Por fim, no exercício **A)**(depósito de agua) proposto não tinha percebido como resolvê-lo em aula, como consequência foi enviado um *mail* de advertência de não estar a cumprir os conteúdos lecionados, durante a semana tinha resolvido o exercício e entreguei com o nome do *package* de `deposito_exercicio_v1` entretanto o docente resolveu o exercício em aula *package* de `deposito_exercicio_professor`

# 5 Conclusão

A Inteligência Artificial é um campo revolucionário que vai se tornar um dos principais focos no futuro desenvolvimento tecnológico. Em um curto espaço de tempo, a Inteligência Artificial deixou de ser apenas uma ideia boa em papel para se tornar uma realidade.

Com isso em mente, os conceitos estudados ao longo desta Unidade Curricular são de extrema importância. A crescente complexidade exigida pelos sistemas mais recentes implica a automação do processo de implementação de software.

Considera-se que nenhuma noção aprendida pode ser vista como menos importante em relação às outras. Esta unidade curricular foi pensada para que os alunos possam adquirir diversas competências em diferentes níveis. Destacam-se competências adquiridas em modularização, fatorização e abstração, bem como a amplicação do conhecimento e a capacidade de identificar padrões e suas possíveis associações com a arquitetura, além de entender a importância disso na escrita de código. Também é essencial compreender as razões por trás dos modelos utilizados.

Com o apoio do docente, considera-se que esta Unidade Curricular permitiu aos alunos melhorar a metodologia de implementação de código. Desde sugestões sobre mecanismos desconhecidos até o uso de facilidades fornecidas pelas linguagens de programação utilizadas, passando por novas noções de simplificação, como encapsulamento, modularidade e eliminação de redundâncias, foram transmitidas boas práticas de programação. Não há dúvida de que essas práticas fornecerão uma base que ajudará os alunos não apenas em outras disciplinas, mas também em suas carreiras profissionais. Essa base serviu como uma transição entre um nível mais básico, focado em fazer as coisas funcionarem, para um nível mais avançado, com a capacidade de desenvolver aplicações consideravelmente mais complexas.

# 6 Bibliografia

# Referências

[1] Prof. Luís Morgado. *Introdução à Inteligência Artificial.* https://2223moodle.isel.pt/pluginfile.php/1197038/mod_resource/content/1/02-introd-ia.pdf

[2] Prof. Luís Morgado. *Introdução à linguagem UML Parte-1.* https://2223moodle.isel.pt/pluginfile.php/1197039/mod_resource/content/2/03-introd-uml-1.pdf

[3] Prof. Luís Morgado. *Introdução à linguagem UML Parte-2.* https://2223moodle.isel.pt/pluginfile.php/1197041/mod_resource/content/1/04-introd-uml-2.pdf

[4] Prof. Luís Morgado. *Modelos de Dinamica.* https://2223moodle.isel.pt/pluginfile.php/1197042/mod_resource/content/2/05-mod-din.pdf

[5] Prof. Luís Morgado. *Introdução à Engenharia de Software.* https://2223moodle.isel.pt/pluginfile.php/1197057/mod_resource/content/1/06-introd-eng-soft.pdf

[6] Prof. Luís Morgado. *Arquitetura de Agentes Reactivos Parte-1* https://2223moodle.isel.pt/pluginfile.php/1197058/mod_resource/content/2/07-arq-react-1.pdf

[7] Prof. Luís Morgado. *Arquitetura de Agentes Reactivos Parte-2.* https://2223moodle.isel.pt/pluginfile.php/1197061/mod_resource/content/2/08-arq-react-2.pdf

[8] Prof. Luís Morgado. *Arquitetura de Agentes Reactivos Parte-3.* https://2223moodle.isel.pt/pluginfile.php/1197063/mod_resource/content/1/09-arq-react-3.pdf

[9] Prof. Luís Morgado. *Procura em Espaços de Estados Parte-1.* https://2223moodle.isel.pt/pluginfile.php/1197236/mod_resource/content/3/11-pee-1.pdf

[10] Prof. Luís Morgado. *Procura em Espaços de Estados Parte-2.* https://2223moodle.isel.pt/pluginfile.php/1204516/mod_resource/content/1/12-pee-2.pdf

[11] Prof. Luís Morgado. *Procura em Espaços de Estados Parte-3.* https://2223moodle.isel.pt/pluginfile.php/1204520/mod_resource/content/1/13-pee-3.pdf

[12] Prof. Luís Morgado. *Arquitetura de Agentes Deliberativos.* https://2223moodle.isel.pt/pluginfile.php/1204525/mod_resource/content/2/14-arq-delib.pdf

[13] Prof. Luís Morgado. *Planeamento Automático com Base em PEE.* https://2223moodle.isel.pt/pluginfile.php/1204527/mod_resource/content/1/15-plan-pee.pdf

[14] Prof. Luís Morgado. *Processo de Decisão Sequencial.* https://2223moodle.isel.pt/pluginfile.php/1207062/mod_resource/content/3/16-pds.pdf

[15] Prof. Luís Morgado. *Planeamento Automático com base em PDM* https://2223moodle.isel.pt/pluginfile.php/1207064/mod_resource/content/3/17-plan-pdm.pdf

[16] Prof. Luís Morgado. *Aprendizagem por Reforço* https://2223moodle.isel.pt/pluginfile.php/1207066/mod_resource/content/3/18-aprend-ref.pdf