Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e Multimédia

Relatório do projeto da disciplina de Inteligência Artificial para Sistemas Autónomos

Docente: Engenheiro Luís Morgado

Aluno: Carlos Simões, nº51696

Turma: 42D

15 de junho de 2025

Índice

Índice de Figuras

Introdução

O projeto é uma iniciativa abrangente que visa desenvolver e implementar sistemas autónomos inteligentes, capazes de interagir com o ambiente e tomar decisões de forma autónoma. Dividido em quatro partes, o projeto aborda desde os fundamentos da arquitetura de sistemas autónomos até técnicas avançadas de planeamento e controlo, passando por modelos reativos e deliberativos.

A primeira parte do projeto introduz a arquitetura base, focando-se nos subsistemas essenciais, como o ambiente e o agente, e na sua interação. Aqui, são explorados conceitos como a perceção do ambiente, a execução de comandos e a modelação de personagens como agentes autónomos. A dinâmica do agente é detalhada através de diagramas de classe e sequência, que ilustram como o agente perceciona o ambiente, processa informações e atua com base em decisões.

Na segunda parte, o projeto avança para esquemas comportamentais reativos, onde são apresentados mecanismos como reações a estímulos, comportamentos compostos e controlo reativo. Esta fase destaca a capacidade do agente de responder a mudanças no ambiente de forma imediata e adaptativa, utilizando hierarquias de comportamentos e prioridades para selecionar ações.

A terceira parte introduz o raciocínio automático com base em modelos de problemas, explorando mecanismos de procura não informada e informada. Aqui, são abordados algoritmos como procura em profundidade, largura e melhor-primeiro, além de técnicas como procura de custo uniforme e avaliação heurística. Esta etapa é crucial para dotar o agente de capacidade de planeamento e resolução de problemas complexos.

Por fim, a quarta parte foca-se no controlo deliberativo, integrando planeamento automático e mecanismos PDM (Processos de Decisão de Markov). O agente é capaz de assimilar informações do ambiente, reconsiderar estratégias e deliberar sobre ações futuras, utilizando modelos do mundo e planeadores para alcançar objetivos de forma eficiente.

Em suma, este projeto oferece uma jornada completa pelo desenvolvimento de sistemas autónomos inteligentes, combinando teoria e prática para criar agentes capazes de operar em ambientes dinâmicos e complexos.

Enquadramento Teórico

Inteligência Artificial e os seus paradigmas

A **Inteligência Artificial** é uma área científica dedicada à criação de sistemas computacionais com capacidade para exibir comportamentos inteligentes. Existem duas abordagens fundamentais no seu desenvolvimento:

- **Analítica**: centra-se na observação empírica, que procura descrever e compreender fenómenos inteligentes a partir da análise de dados e padrões observáveis;
- **Sintética**: baseia-se em modelos teóricos para conceber sistemas que simulem ou reproduzam comportamentos inteligentes, mesmo que não correspondam exatamente ao funcionamento da inteligência humana

Esta área assenta em estruturas concetuais compostas por conjuntos de ideias, modelos e princípios que orientam a análise de problemas e a formulação das respetivas soluções. Estas estruturas são designadas por **paradigmas**, sendo possível identificar três paradigmas fundamentais no domínio da IA:

- **Simbólico**: baseia-se na hipótese de que o sistema físico de símbolos é suficiente para a inteligência. Esta resulta de processos computacionais aplicados a estruturas simbólicas;
- **Conexionista**: defende que a **inteligência** emerge das interações entre um grande número de **unidades elementares de processamento** interligadas – os chamados **neurónios**. Nestes sistemas, a **informação é representada e processada de forma distribuída**. Assim, o conhecimento não está armazenado em símbolos explícitos, mas sim na **configuração e força das conexões** dentro da rede;
- **Comportamental**: este paradigma acredita que a inteligência surge do **comportamento do sistema em resposta a estímulos**, manifestando-se através da sua **capacidade de adaptação**. Este princípio é aplicado a agentes autónomos que interagem dinamicamente com o ambiente que os rodeia

Representação do conhecimento

A representação do conhecimento configura-se como um dos pilares fundamentais da Inteligência Artificial (IA), ao possibilitar que sistemas computacionais **armazenem**, **organizem** e **processem informações** de modo estruturado, servindo de base para a **realização de inferências** e a **tomada de decisões**. Este domínio de estudo **examina os mecanismos através dos quais dados brutos são convertidos em conhecimento relevante** para a IA, com ênfase nos **três níveis hierárquicos de representação** e no conceito de **ancoragem simbólica**.

Os três níveis de hierarquia são:

- **Dados**: conjunto de símbolos ou sinais que representam observações ou medições relativas a um domínio específico, ainda desprovidos de significado por si só;
- **Informação**: conjunto de dados organizados de forma estruturada que adquirem significado quando interpretados num determinado contexto;
- **Conhecimento**: conjunto articulado de informações inter-relacionadas que permite a realização de inferências e a tomada de decisões dentro de um domínio específico

A ancoragem simbólica é a relação que estabelece o significado das representações simbólicas, uma vez que estas **não possuem sentido intrínseco**. O significado emerge das interações entre as **estruturas simbólicas** e da correspondência destas com as **entradas** e **saídas** do **sistema**, isto é, com o **domínio de referência**. Assim, os símbolos adquirem significado ao serem ancorados em observações e conceitos do mundo real.

Para implementar a representação do conhecimento, a IA utiliza estruturas como:

- **Redes Semânticas**: são **grafos** (estruturas de nós e arestas) utilizados para representar conhecimento de forma visual e hierárquica;
- **Ontologias**: **representações formais** e **estruturadas** do conhecimento sobre um domínio específico;
- **Lógica**: sistema formal para representar conhecimento e **realizar inferências através de regras** bem definidas

Sistemas Autónomos

Um sistema autónomo inteligente funciona com base num ciclo de retroalimentação composto pelas fases de perceção, processamento e ação. Este ciclo permite o controlo contínuo das funções do sistema, orientando o seu comportamento no sentido de alcançar os objetivos estabelecidos.

Ambientes

Em Inteligência Artificial, os ambientes definem os contextos onde os agentes inteligentes operam, estabelecendo as regras, restrições e propriedades que influenciam a perceção, o processamento e a ação dos mesmos.

A sua classificação é fundamental para a escolha de técnicas adequadas — como a lógica simbólica em ambientes determinísticos ou o aprendizado por reforço em ambientes estocásticos.

Nestes contextos, a representação do ambiente desempenha um papel crucial no suporte ao processamento interno do agente, podendo assumir diferentes níveis de complexidade, frequentemente recorrendo a estruturas como grafos.

No caso de ambientes físicos contínuos, é necessária a discretização das perceções para possibilitar o seu tratamento computacional.

Agentes Inteligentes

Um agente inteligente é a representação computacional de um sistema autónomo inteligente, que opera num ambiente segundo o ciclo perceção-processamento-ação, onde:

- **Perceção**: obter dados do ambiente;
- **Processamento**: análise das informações obtidas e tomada de decisões;
- **Ação**: executar tarefas

Este tem de ser **autónomo**, ou seja, operar sem a intervenção de outros sistemas; **reativo**, conseguindo responder às mudanças num ambiente em tempo real (**estímulos**); **pró-ativo**, isto é, capaz de tomar iniciativa e cumprir objetivos; e por fim, tem de ser sociável, mais concretamente, tem de ser capaz de interagir com outros agentes num mesmo ambiente partilhado.

Cognição e Racionalidade

A **cognição** diz respeito aos processos de aquisição, processamento e utilização de informação por sistemas inteligentes, inspirando-se no funcionamento do cérebro humano. Este conjunto de capacidades permite que os agentes percecionem, aprendam e se adaptem ao ambiente em que operam.

Por sua vez, a **racionalidade** refere-se à aptidão para tomar decisões ótimas com base em objetivos previamente definidos, procurando maximizar a eficácia da ação, tanto em contextos controlados (e.g. jogos), como em ambientes incertos e dinâmicos (e.g. veículos autónomos).

Enquanto a cognição fornece os mecanismos necessários à compreensão e interação com o mundo, a racionalidade assegura que as decisões tomadas sejam lógicas e orientadas para metas específicas.

Em conjunto, estas capacidades constituem a base do funcionamento dos agentes inteligentes — desde assistentes virtuais até robôs autónomos — promovendo um equilíbrio entre processamento avançado de informação e tomada de decisão estratégica.

Arquiteturas de agente

As arquiteturas de agente definem a estrutura interna dos sistemas inteligentes, determinando a forma como estes percecionam o ambiente, processam informação e atuam sobre o mesmo. Estas arquiteturas integram componentes de reatividade,

deliberação e aprendizagem, de modo a orientar o comportamento do agente na prossecução de objetivos específicos.

Os três modelos existentes são:

- **Reativo**
  - Paradigma: comportamental;
  - Funcionamento: baseado em associações diretas entre estímulos e respostas sem representação interna;
  - Finalidade: objetivos subentendidos;
- **Deliberativo**
  - Paradigma: simbólico;
  - Funcionamento: utiliza raciocínio e tomada de decisão com representações internas e objetivos expressos;
  - Finalidade: objetivos claros;
- **Híbrido**
  - Funcionamento: combina elementos reativos e deliberativos, permitindo uma resposta eficiente e ao mesmo tempo fundamentada;
  - Finalidade: objetivos explícitos

Introdução à engenharia de software

A engenharia de software constitui uma área da Engenharia que se dedica ao estudo e aplicação de princípios e metodologias sistemáticas, disciplinadas e mensuráveis para o desenvolvimento, operação e manutenção de sistemas de software com qualidade.

Alguns aspetos fundamentais da engenharia de software são:

- **Sistemática**: refere-se à organização metódica dos processos, promovendo previsibilidade e assegurando o cumprimento rigoroso dos requisitos funcionais e não funcionais;
- **Quantificável**: implica a capacidade de medir objetivamente os processos e os produtos de software, com o intuito de garantir níveis aceitáveis de qualidade, desempenho e fiabilidade

Por fim, a engenharia de software também tem os seus desafios, sendo os principais:

- **Complexidade**: a crescente sofisticação e dimensão dos sistemas tornam a sua compreensão e gestão progressivamente mais difíceis, exigindo soluções robustas;
- **Mudança**: a necessidade constante de adaptação a novos requisitos de utilizadores e à rápida evolução tecnológica impõe desafios significativos à manutenção e progresso dos sistemas

Complexidade e arquitetura de software

À medida que os sistemas de software se tornam mais sofisticados, cresce também a sua complexidade intrínseca, tornando essencial a adoção de princípios arquitetónicos sólidos para garantir a sua viabilidade a longo prazo. A arquitetura de software surge, assim, como uma disciplina fundamental para organizar e estruturar sistemas de forma a reduzir a complexidade, melhorar a manutenção e facilitar a evolução. Compreender os diferentes tipos de complexidade e aplicar estratégias como a abstração, modularização e encapsulamento permite não só simplificar o desenvolvimento, mas também promover a qualidade, escalabilidade e robustez das soluções desenvolvidas.

Que tipos de complexidade existem?

- **Complexidade organizada**: resulta da existência de padrões bem definidos nas relações entre os componentes do sistema, permitindo uma maior previsibilidade;
- **Complexidade desorganizada**: caracteriza-se por interações irregulares entre componentes diferentes, dificultando a análise e a manutenção do sistema

Para reduzir a complexidade de um sistema, foram adotadas várias técnicas, tais como:

- **Abstração**: processo de focar nos aspetos essenciais de um sistema, simplificando o mesmo através da omissão de detalhes irrelevantes, com o objetivo de facilitar a sua compreensão, modelação e desenvolvimento;
- **Modularização**: divisão do sistema em módulos coesos, de forma a facilitar a implementação, a manutenção e o reuso de componentes;
- **Decomposição**: fragmentação funcional ou estrutural de um sistema em partes menores e mais gerenciáveis;
- **Encapsulamento**: isolamento dos detalhes de implementação, promovendo o princípio da ocultação de informação;
- **Factorização**: identificação e remoção de redundâncias no sistema, recorrendo a mecanismos como a herança ou a delegação

As métricas de arquitetura constituem instrumentos fundamentais para avaliar a qualidade estrutural de um sistema de software, permitindo a analisar a eficácia da sua modularização e a facilidade de manutenção e evolução (adaptado de *Software Engineering: 10th Edition, Ian Sommerville*, 2016, p. 168). As principais métricas são:

- **Acoplamento**: mede o grau de interdependência entre módulos – quanto menor, melhor a modularidade;
- **Coesão**: refere-se ao grau de relacionamento entre os elementos internos de um módulo - quanto maior, melhor a qualidade funcional;
- **Simplicidade e Adaptabilidade**: avaliam a facilidade de compreensão, modificação e evolução da arquitetura do sistema;

Modulação com UML (Unified Modeling Language)

A UML fornece um conjunto de notações padronizadas destinadas à modelação de sistemas orientados a objetos, facilitando a comunicação entre os intervenientes no processo de desenvolvimento.

Os diagramas UML permitem representar graficamente a estrutura e comportamento de sistemas, facilitando a sua compreensão e comunicação. Os diagramas mais relevantes são:

- **Estruturais:**
  - **Diagrama de classes**: representa a estrutura do sistema;
- **Comportamentais**:
  - **Diagrama sequencial**: representa interações temporais entre objetos;
  - **Diagrama de estados (máquinas de estados)**: modela a dinâmica de estados de um objeto;
  - **Diagrama de atividades**: descreve fluxos de trabalho

Os elementos principais dos UMLs são:

- **Classes**: incluem atributos, operações e controlos de visibilidade (público, privado, protegido);
- **Relações**: associação, agregação, composição, generalização e dependência entre classes e objetos;
- **Interfaces**: definem contratos funcionais independentes da implementação específica

Dinâmica de Sistemas e Máquinas de Estados

A dinâmica de sistemas foca-se no comportamento dos mesmos ao longo do tempo, sendo frequentemente modelada através de máquinas de estados, que descrevem as possíveis configurações do sistema e as transições entre elas em resposta a eventos. Esta abordagem permite analisar e implementar sistemas reativos de forma rigorosa e previsível.

Um **estado** representa uma configuração específica do sistema num determinado momento, captando toda a informação relevante sobre a sua condição interna. É uma abstração que descreve a situação atual do sistema em relação ao seu comportamento e características.

Uma **transição** é a mudança de um estado para outro, desencadeada por um evento específico, podendo estar associada a uma ação que o sistema executa durante a mudança. Para que haja uma transição, é necessário:

- **Evento** – ocorrência que desencadeia a transição;
- **Estado de origem** – estado atual antes da transição;

- **Estado de destino** – novo estado após transição;
- **Ação** – caso seja uma função de saída, a ação é o comportamento executado durante a transição

A função de transição de estado pode ser definida da seguinte maneira:

$$\delta : Q \times \Sigma \rightarrow Q,$$

onde Q é o conjunto de estados e Σ é o alfabeto de entrada, ou seja, os eventos

Já a função de saída, pode ser formalmente escrita como:

Máquinas de Mealy: a função de saída depende das entradas

$$\lambda : Q \times \Sigma \rightarrow Z$$

ou

Máquinas de Moore: a função de saída não depende das entradas

$$\lambda : Q \rightarrow Z$$

onde Q é o conjunto de estados e Σ é o alfabeto de entrada, ou seja, os eventos, e Z é o alfabeto de saída (ações possíveis)

Uma **máquina de estados** finita (*finite state machine*) é um modelo matemático usado para descrever o comportamento de sistemas através de um conjunto finito de estados, transições e ações, em que o sistema se encontra num único estado de cada vez e muda de estado em resposta a eventos ou condições específicas, permitindo assim, representar de forma estruturada processos de decisão e controlo em contextos como automação, jogos ou sistemas reativos.

Projeto

Ao longo do semestre, foi realizado um projeto com o objetivo de aplicar, de forma prática, os conceitos teóricos abordados nas aulas. A seguir, apresenta-se a descrição detalhada de cada etapa do projeto, articulando a componente prática com os fundamentos teóricos subjacentes.

Parte 1

Nesta fase inicial do projeto, procurou-se implementar um sistema baseado em agentes inteligentes, recorrendo a uma máquina de estados finita para modelar o comportamento de uma personagem num ambiente interativo de jogo.

A personagem foi concebida para interagir com o meio envolvente através de um ciclo de perceção, processamento e ação. O ambiente simula uma evolução contínua ao gerar eventos que são percecionados pela personagem. Com base nesses estímulos, a máquina de estados determina a transição de estado mais adequada, definindo a ação subsequente e ajustando o comportamento da personagem em conformidade.

Esta abordagem tem como principal finalidade simular uma interação dinâmica e adaptativa entre a personagem e o ambiente virtual, explorando conceitos fundamentais da inteligência artificial aplicada e princípios de engenharia de software.

Subsistema ambiente

O subsistema **Ambiente** constitui um elemento central na arquitetura de sistemas autónomos, sendo desenvolvido em conformidade com os princípios de engenharia de software abordados nas aulas, e com a matéria acerca de inteligência artificial e sistemas autónomos lecionada.

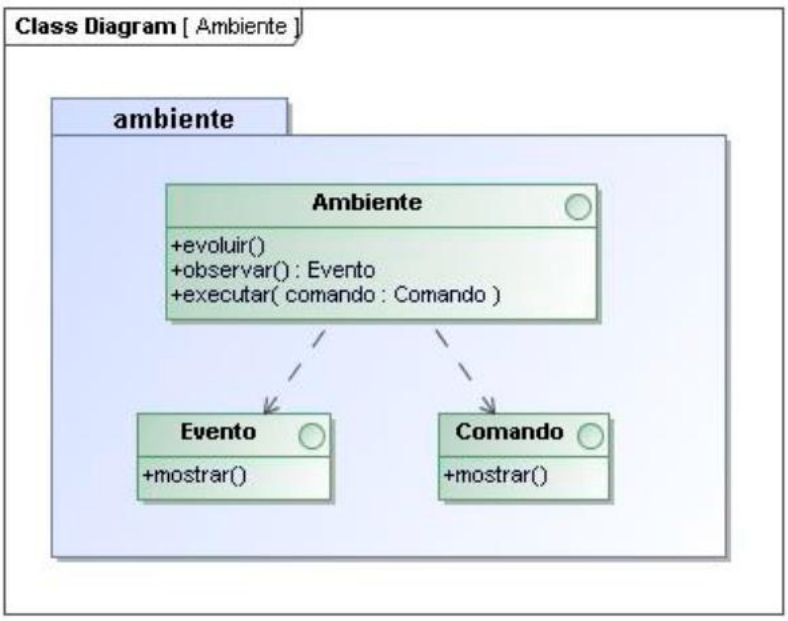


Figura 1 - Diagrama de classes do package ambiente

Este subsistema designa o meio – físico ou virtual (neste caso é virtual) – com o qual o agente interage, sendo composto por três elementos essenciais: o **ambiente**, entendido como o domínio envolvente onde ocorrem alterações relevantes e onde o agente poderá atuar; o **evento**, definido como qualquer mudança percetível nesse ambiente, captada por sensores e capaz de influenciar o comportamento do agente; e o **comando**, que corresponde à ação gerada pelo agente em resposta aos eventos detetados, transmitida a atuadores com o intuito de modificar ou influenciar o ambiente. O subsistema é, assim, fundamental para a articulação entre perceção e atuação, garantindo a autonomia e adaptabilidade do sistema.

O mesmo é então constituído por três interfaces principais que formam um contrato para a interação entre agente(s) e o(s) ambiente(s):

1. **Ambiente** – interface principal que define as operações básicas do ambiente;
2. **Evento** – representa ocorrências no ambiente que o agente pode perceber;
3. **Comando** – representa ações que o agente pode executar no ambiente

Como foi mencionado, tanto nesta primeira parte como no resto do projeto, foram aplicados princípios de engenharia de software, tais como:

- **Abstração** – as interfaces disponibilizam uma visão genérica do ambiente, ocultando detalhes da sua implementação. Esta abordagem permite múltiplas concretizações adaptadas a diversos cenários, neste caso, ao ambiente de jogo;

- **Modularização** – o ambiente é estruturado como um subsistema autonomamente delimitado, com responsabilidades claramente definidas. Apresenta um baixo grau de acoplamento relativamente a outros subsistemas, nomeadamente o agente;
- **Encapsulamento** – os detalhes internos do funcionamento do ambiente são ocultados, sendo expostos unicamente através de interfaces bem definidas, promovendo a integridade e a segurança da arquitetura

Subsistema agente e dinâmica do agente

O subsistema **agente**, segundo o diagrama de classes, é responsável por modelar o comportamento inteligente dentro do sistema. Este representa o agente autónomo que interage com o ambiente, percecionando-o e atuando sobre ele.

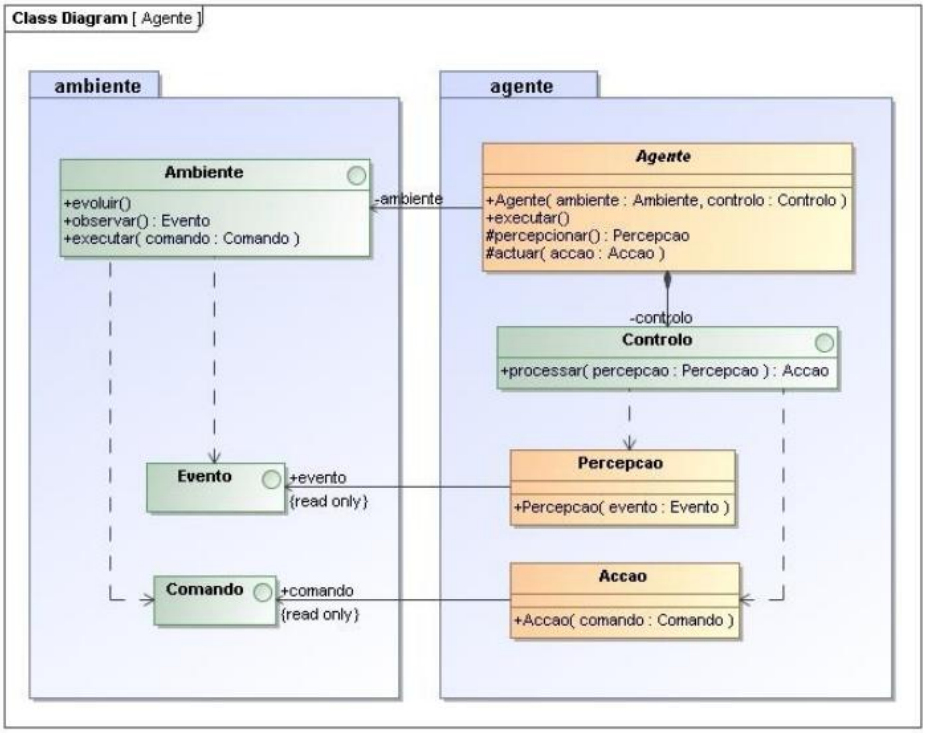


Figura 2 - Diagrama de classes do package agente

Este é composto por:

1. **Agente** – classe abstrata que representa um agente autónomo. Um agente autónomo é uma entidade que interage com um ambiente, operando num ciclo de perceção, ou seja, observar e recolher informações; processamento, onde o agente processa as mesmas e decide a ação a realizar; e ação, que é ação mais indicada perante as informações recolhidas e processadas;

2. **Controlo** – interface que processa as perceções do agente, resultando em ações que o mesmo irá executar, tendo em conta a sua situação e o estado do ambiente;
3. **Percepcao** – classe que representa o que foi interpretado pelo agente a partir de um evento observado no ambiente;
4. **Accao** – classe que reflete o resultado do processamento de uma perceção, que é a ação a executar face à decisão que o mesmo tomou

Aqui foram aplicados princípios da engenharia de software que contribuem para a qualidade e robustez do sistema. A **abstração** é evidenciada através do uso da interface *Controlo*, e da classe abstrata *Agente*, que definem contratos genéricos e permitem múltiplas implementações sem expor os detalhes internos. O **encapsulamento** está presente na proteção dos atributos privados, como *evento* em *Percepcao*, *comando* em *Accao* e *controlo* em *Agente*, sendo estes acedidos apenas por métodos públicos (getters), assegurando a integridade do estado interno das classes. A **modularização** é garantida pela clara separação de responsabilidades entre as classes. O princípio da **responsabilidade** é respeitado, com cada classe e método a desempenhar um papel bem definido, evitando sobreposição de funcionalidades. A **extensibilidade,** neste contexto, refere-se à facilidade de adicionar novas funcionalidades ao sistema sem modificar o código existente. Por fim, a **reutilização** é promovida pela conceção de componentes genéricos, como *Accao*, *Percepcao* e *Comando*, que podem ser facilmente integrados em diferentes partes do sistema. Em conjunto, estes princípios tornam o sistema mais robusto, modular, fácil de manter e preparado para futuras evoluções.

Subsistema ambiente do jogo

O subsistema **ambiente de jogo** constitui uma especialização do subsistema ambiente, concebida para modelar as regras, eventos e comandos específicos de um determinado jogo, neste caso, para um jogo de simulação de procura, aproximação, fotografia e observação de animais no seu ambiente.

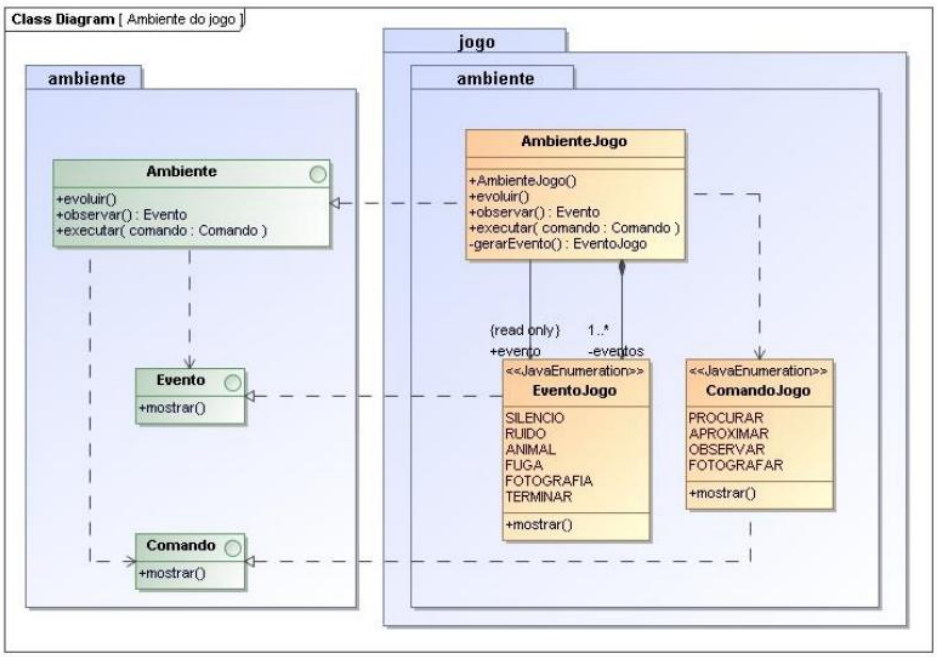


Figura 3 - Diagrama de classes do package jogo.ambiente

O subsistema é organizado em três classes:

1. **AmbienteJogo** - implementa a interface Ambiente. Gere o estado do ambiente, mantém o evento atual (EventoJogo), permite ao agente observar o ambiente (retornando o evento atual) e executar comandos. Usa um *Map* para associar *inputs* do utilizador a eventos do jogo;
2. **EventoJogo** - enumerado que implementa a interface Evento. Representa todos os tipos de eventos possíveis no ambiente do jogo, que são silêncio, ruído, animal, fuga, fotografia e terminar. Cada evento pode ser mostrado no ecrã;
3. **ComandoJogo** - enumerado que implementa a interface Comando. Define todos os comandos que o agente pode executar no ambiente do jogo: procurar, aproximar, observar e fotografar. Qualquer um dos comandos, pode ser apresentado no ecrã.

Os princípios de engenharia de software aqui presentes, são os mesmos do subsistema anteriormente falado, mas dirigidos às classes competentes do ambiente de jogo.

Resumindo, o ambiente de jogo gera eventos a partir dos comandos do utilizador, que o agente pode percecionar, e posteriormente, atuar. Os enumerados presentes garantem que o sistema apenas reconhece os eventos e comandos pré-definidos, ou seja, eventos e comandos válidos.

Subsistema e controlo da personagem

O subsistema **personagem** corresponde à componente responsável por modelar a lógica e o comportamento da entidade controlada pelo agente no contexto do jogo. Uma personagem é um agente com comportamentos, perceções e ações adaptadas ao ambiente específico do jogo.

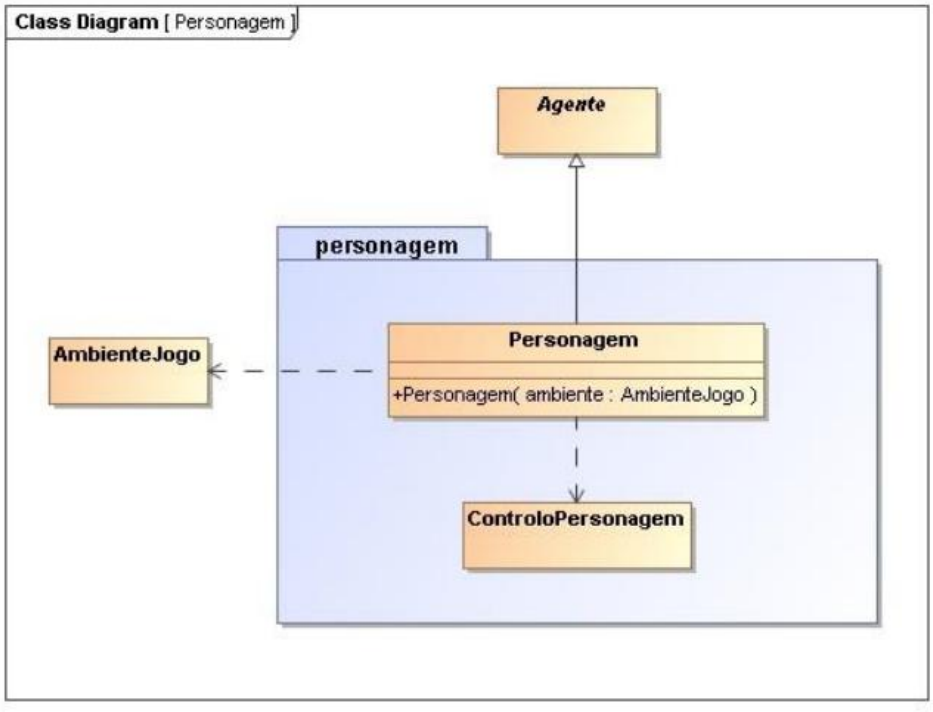


Figura 4 - Diagrama de classes do subsistema personagem

Este subsistema tem como componentes Agente e AmbienteJogo (já falados anteriormente), e Personagem e ControloPersonagem (que será falado posteriormente):

- Classe Personagem
    - Herda de agente, ou seja, possui todas as capacidades básicas de um agente (percecionar, decidir e atuar), e é adaptada ao contexto do ambiente de jogo, pois o seu construtor recebe-o como argumento

No que toca ao **controlo da personagem**, este é responsável por decidir, de forma autónoma, as ações que a personagem deve executar no jogo, com base nas perceções recebidas do ambiente. Este implementa a interface Controlo para sistematizar o comportamento.

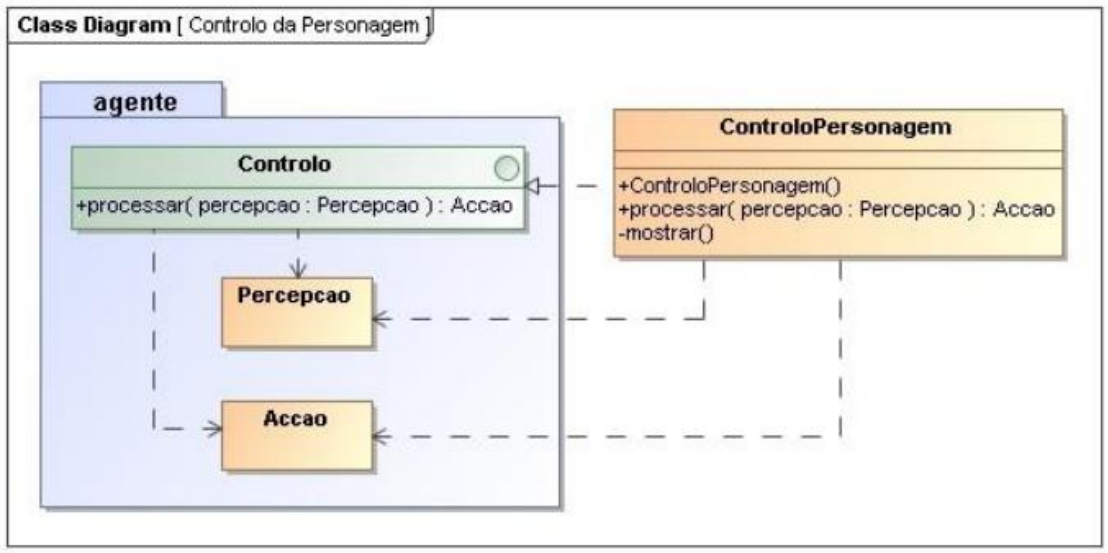


Figura 5 - Diagrama de classes do ControloPersonagem

A classe ControloPersonagem foi então implementada da seguinte maneira:

1. **ControloPersonagem()** – cria todos os estados e define as transições e ações associadas a cada evento;
2. **Processar(percepcao : Percepcao)** – recebe uma perceção, extrai o evento, processa-o e retorna a ação a executar;
3. **Mostrar()** – exibe o estado atual da personagem

É percetível a presença de abstração nesta *package*, pois ControloPersonagem é implementada a partir da interface Controlo. Existe também a reutilização, devido à integração das classes Percepcao e Accao; e por fim, está presente o conceito de modularização, pois ControloPersonagem está separado de Personagem, criando um módulo com a sua própria responsabilidade.

Em suma, o controlo de personagem permite que a personagem reaja de forma estruturada e previsível aos eventos do ambiente.

Subsistema jogo

O subsistema **jogo** é o núcleo que coordena toda a execução do sistema, integrando as várias componentes desenvolvidas anteriormente, para criar o ciclo completo de funcionamento do jogo. Este define como o jogo é iniciado, como o agente interage com a ambiente, e como a simulação decorre ao longo do tempo.

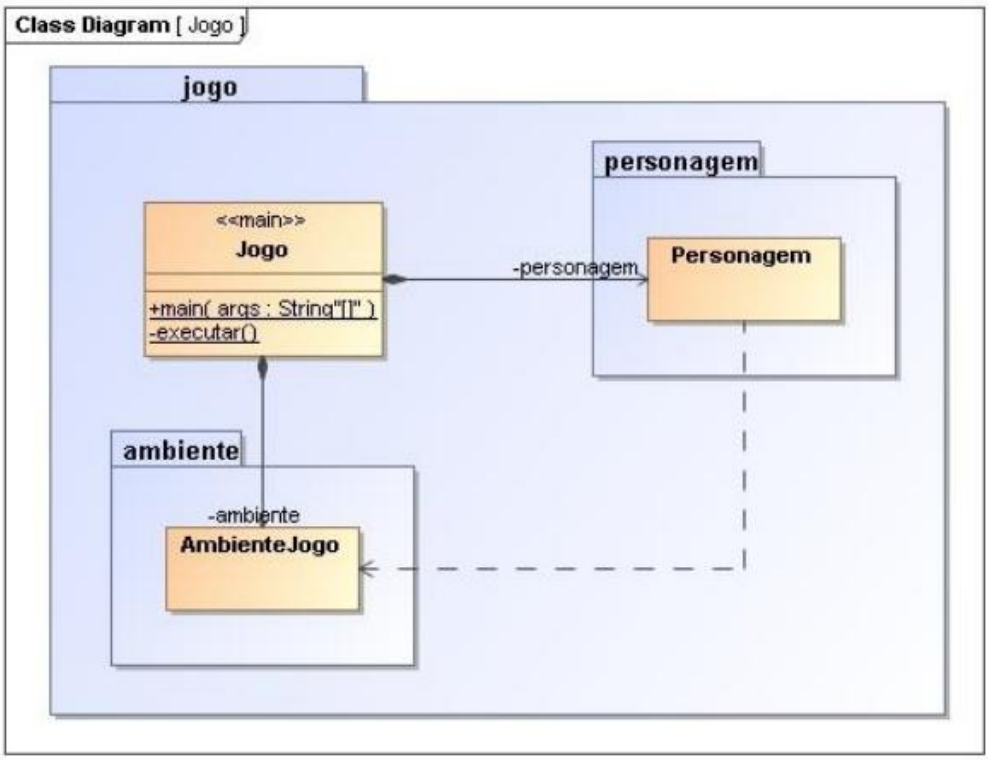


Figura 6 - Diagrama de classes da package jogo

A classe Jogo integra por composição uma instância de Personagem e outra de AmbienteJogo (classes anteriormente explicadas), e a sua implementação contém o método ***main***, que é o ponto de partida do programa. Nele são criados dois objetos principais – um ambiente (instância de AmbienteJogo), que representa o mundo do jogo, e uma personagem (instância de Personagem), que representa o agente que interage com o ambiente.

O ciclo principal de execução foi implementado a partir do seguinte diagrama sequencial:

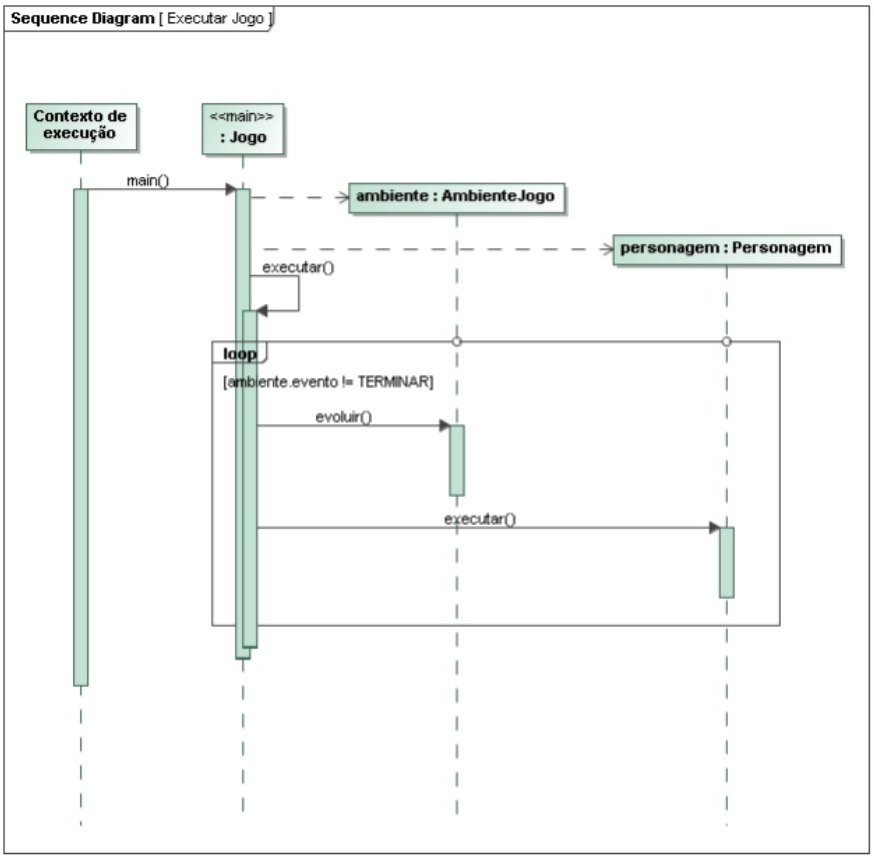


Figura 7 - Diagrama sequencial do ciclo de execução do jogo

Enquanto o evento do ambiente não for TERMINAR (ou seja, enquanto ambiente.evento != TERMINAR), o sistema entra num ciclo em que o ambiente evolui e a personagem atua. Neste ciclo, o método evoluir() da classe *AmbienteJogo* é invocado, atualizando o estado do mundo. De seguida, é chamado o método executar() da *Personagem*, no qual esta observa o ambiente, decide a ação a realizar através do seu módulo de controlo e, por fim, atua, enviando os comandos apropriados ao ambiente.

Subsistema Máquina de Estados

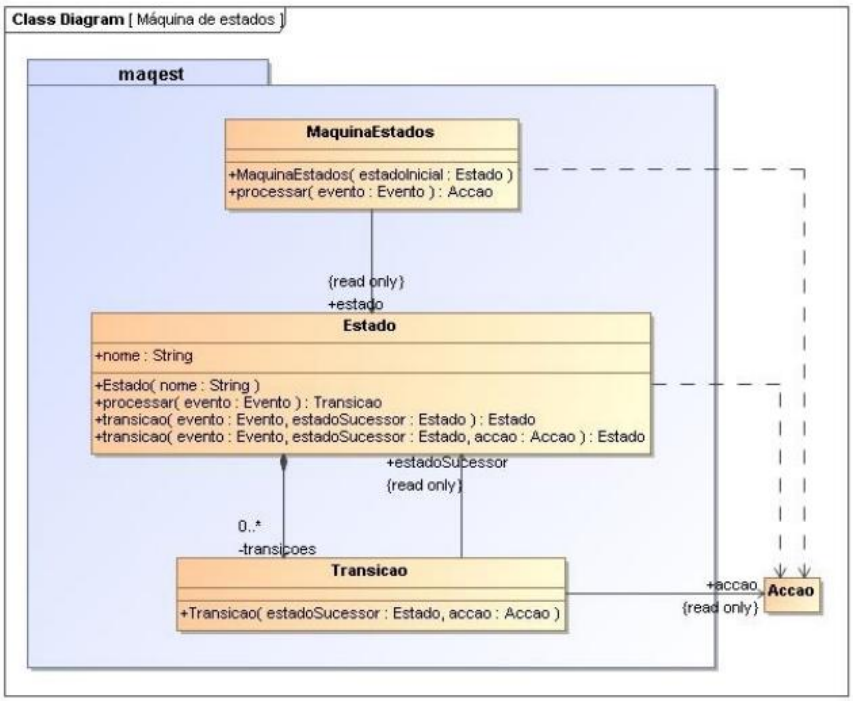


Figura 8 - Diagrama de classes da package maqest

Este subsistema é constituído por três classes principais, sendo elas:

1. **Estado** – guarda o nome do estado e um mapa de transições (evento -> transição). Permite definir transições para outros estados, com (função de saída) ou sem ação associada (função de transição de estado). Também permite processar um evento e obter a transição correspondente;
2. **Transicao** – representa uma transição entre estados. Guarda o estado sucessor e a ação a executar;
3. **MaquinaEstados** – mantém o estado atual. Processa eventos, ou seja, verifica se existe transição para o evento, atualiza o estado e retorna a ação associada

Em síntese, o subsistema permite modelar comportamentos complexos de forma estruturada, facilitando a manutenção e evolução do sistema.

O método processar() da classe MaquinaEstados foi realizado segundo o seguinte diagrama:

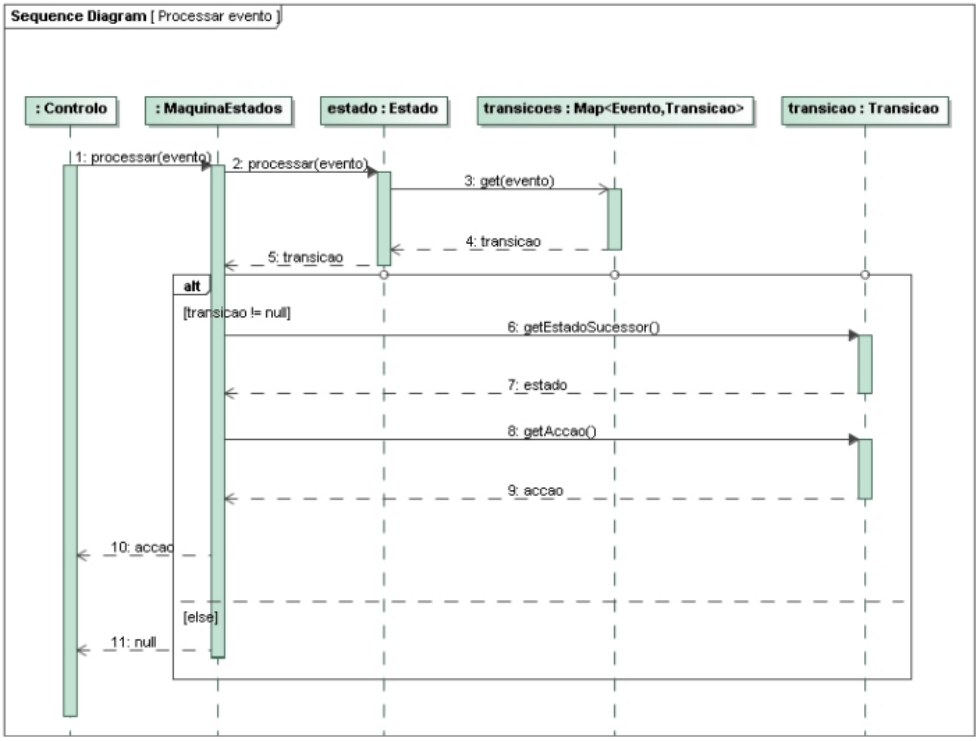


Figura 9 - Diagrama sequencial para implementação do método processar(evento : Evento) da classe MaquinaEstados

O diagrama sequencial indica que para cada evento, é verificada a existência de uma transição, dá-se uma mudança de estado se necessário e retorna a ação a executar, caso haja uma transição aplicável.

Controlo da personagem atualizado e dinâmica da personagem

Devido à inclusão de uma máquina de estados, foi necessário atualizar o controlo da personagem, de modo que ficasse em conformidade com o diagrama de classes remodelado.

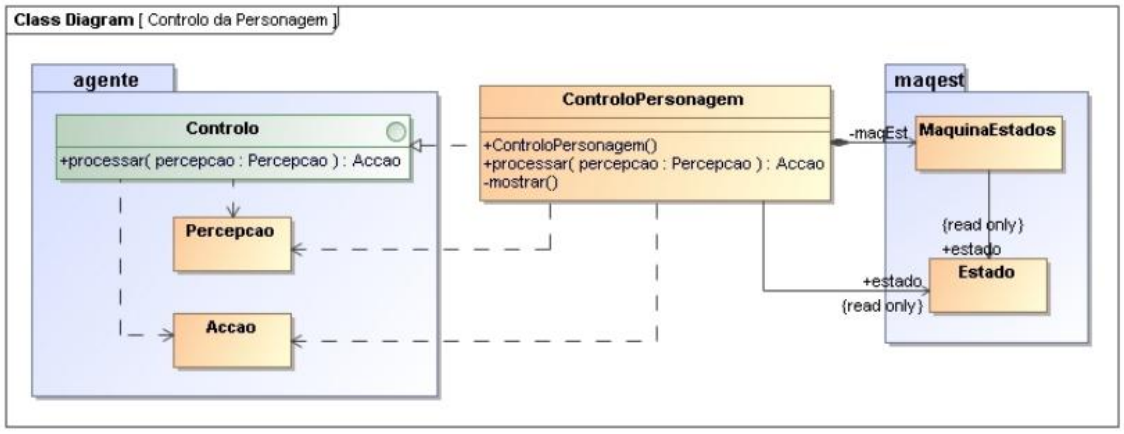


Figura 10 - Diagrama de classes do controlo da personagem modificado

Tirando a *package* agente, a este controlo foi incluída a *package* maqest, de modo a ser possível incorporar uma máquina de estados para gerir o comportamento da personagem.

Adicionou-se então, no construtor de ControloPersonagem, a definição de todos os estados, ações e transições possíveis e válidas.

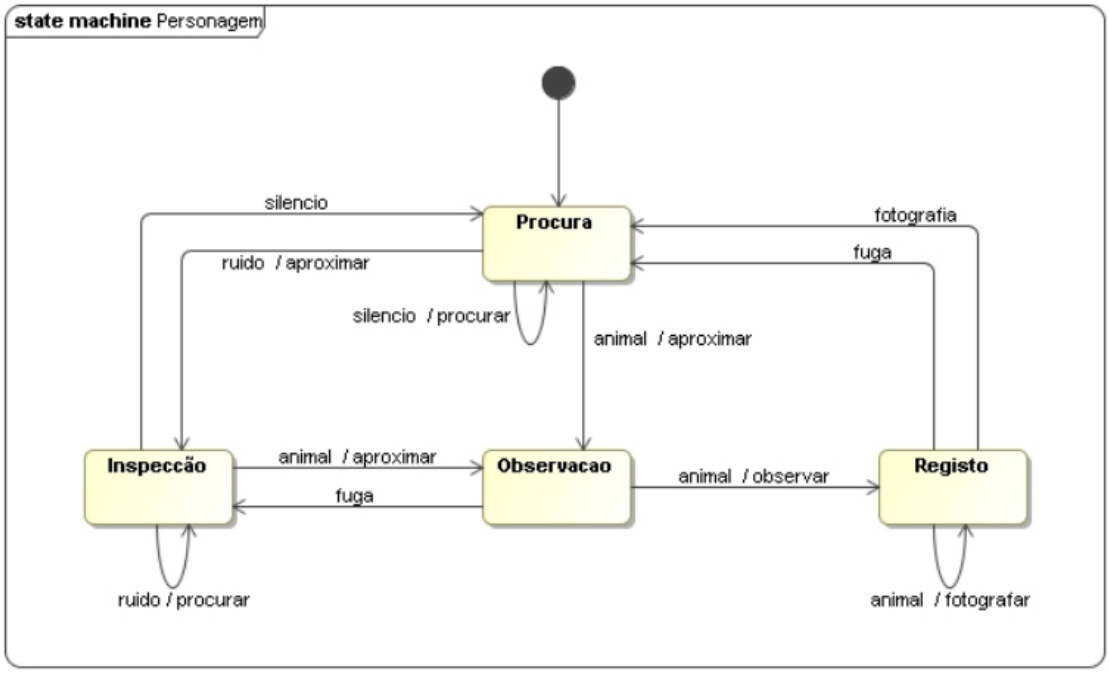


Figura 11 - Máquina de estados incorporada no ControloPersonagem

Foi, por fim, necessário modificar o método processar(percepcao : Percepcao) de modo a incluir a máquina de estados.

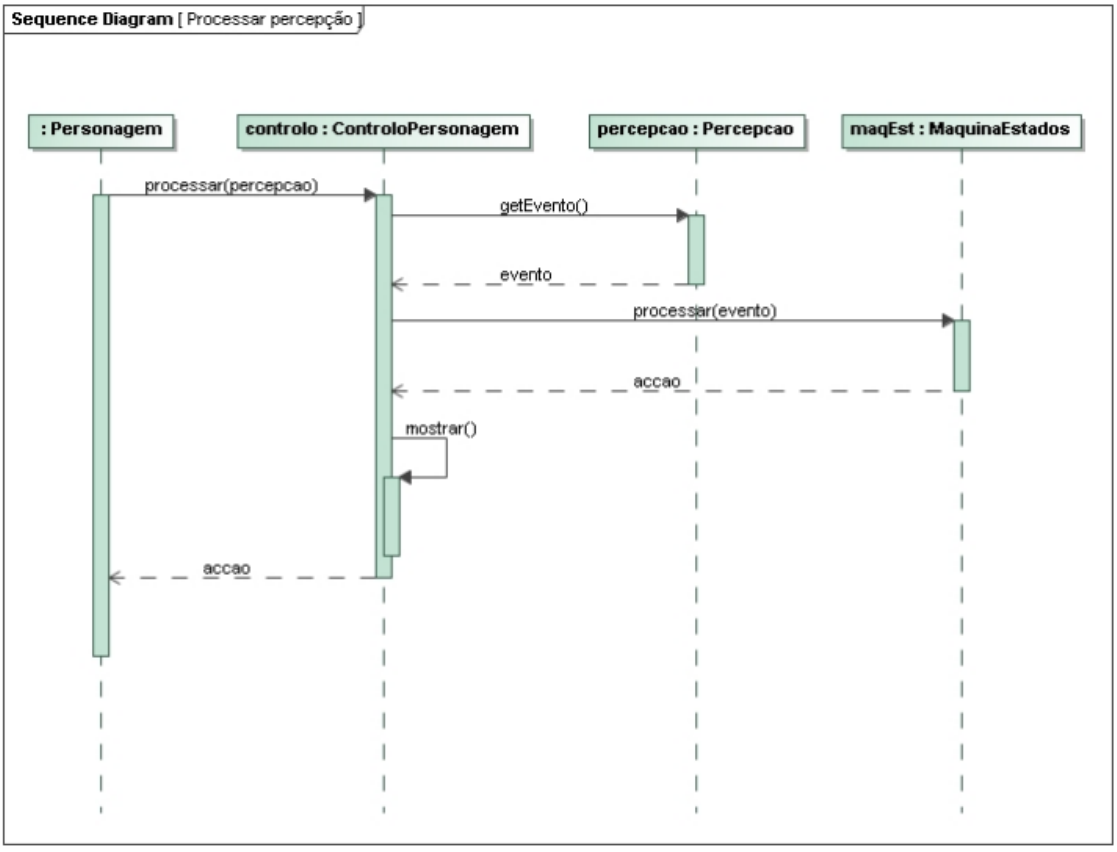


Figura 12 - Diagrama sequencial para implementação comportamental do método processar(percepcao : Percepcao)

O método passa então a obter o evento da perceção recebida, encaminhando-o para a máquina de estados, que determina a ação adequada. Em seguida, chama o método mostrar() para exibir o estado atual, e finalmente devolve a ação resultante do processamento da perceção.

```
Evento? a

Evento: ANIMAL
Estado atual: Observação
Comando: APROXIMAR

Evento? f

Evento: FUGA
Estado atual: Inspecção

Evento? r

Evento: RUIDO
Estado atual: Inspecção
Comando: PROCURAR

Evento? a

Evento: ANIMAL
Estado atual: Observação
Comando: APROXIMAR

Evento? o

Evento: FOTOGRAFIA
Estado atual: Observação

Evento? t

Evento: TERMINAR
Estado atual: Observação
```

Figura 13 - Exemplo de output da classe Jogo

Este output resulta do ciclo de interação entre o ambiente e a personagem, onde cada evento introduzido pelo utilizador provoca uma transição de estado e uma ação (comando), de acordo com a máquina de estados que foi definida em ***ControloPersonagem***. O output mostra sempre o evento, o estado atual e, se existir, o comando associado à ação.

Parte 2

Esta parte do projeto foca-se na conceção e implementação de uma arquitetura reativa para um agente autónomo capaz de navegar num ambiente discreto com obstáculos e alvos. O objetivo principal é desenvolver um sistema que, de forma modular, concretize comportamentos como exploração, aproximação de alvos e evitamento de obstáculos, utilizando associações estímulo-resposta. Baseada nos princípios das arquiteturas reativas, descritos nos documentos fornecidos pelo docente, a arquitetura proposta organiza-se em elementos como ***Reaccao***, ***Estimulo***, ***Resposta*** e ***Comportamento***, com ênfase em comportamentos simples e compostos. Estes são coordenados por um mecanismo de controlo reativo, que processa perceções para gerar ações sem representações internas complexas, conforme ilustrado nos diagramas de classes e sequência.

Arquitetura de Agentes Reativos

I. Conceitos Fundamentais dos Agentes Reativos

Os agentes reativos constituem uma classe de sistemas cujo comportamento emerge de forma imediata e automática, resultando de associações diretas entre **estímulos sensoriais** (**perceções do ambiente**) e **respostas** motoras (**ações**). Nestes agentes, os objetivos não se encontram representados de forma explícita ou simbólica, estando antes incorporados nos próprios mecanismos de mapeamento estímulo-resposta que orientam a sua atuação. Caracterizam-se por um forte acoplamento ao ambiente, frequentemente implementado através de ligações diretas entre sensores e atuadores. Um exemplo paradigmático desta abordagem encontra-se nos Veículos de *Braitenberg*, concebidos como modelos experimentais para explorar como comportamentos aparentemente complexos podem emergir de estruturas simples, onde estímulos como luz ou proximidade de obstáculos ativam respostas motoras sem recurso a representações internas do meio.

O comportamento de um agente reativo estrutura-se em torno de um ciclo perceção–reação–ação, em que as reações são organizadas em módulos comportamentais. Estes módulos podem assumir a forma de comportamentos simples, frequentemente descritos por regras do tipo *SE-ENTÃO*, ou de comportamentos compostos, que integram múltiplos subcomportamentos. Por exemplo, um agente concebido para recolher alvos num ambiente dinâmico poderá integrar comportamentos como *AproximarAlvo*, *EvitarObstáculo* e *Explorar*, estruturados hierarquicamente de forma a permitir uma resposta adaptativa a diferentes contextos ambientais.

II. Mecanismos de Reação e Coordenação de Comportamentos

Num agente reativo, uma perceção pode ativar múltiplas reações, o que levanta a questão de como selecionar a ação a executar. Existem três mecanismos principais para essa seleção:

1. **Execução Paralela de Ações**: ações que não interferem entre si podem ser executadas em simultâneo, desde que a infraestrutura do agente o permita (por exemplo, através de múltiplos atuadores independentes);
2. **Prioridade de Ações**: quando as ações são conflituosas, é atribuída uma prioridade a cada uma, sendo selecionada aquela com maior prioridade. Por exemplo, evitar um obstáculo pode ter prioridade sobre avançar na direção de um alvo;
3. **Combinação de Ações**: ações distintas podem ser combinadas numa única ação resultante, como no caso da soma vetorial de direções de movimento

A coordenação de comportamentos revela-se essencial em agentes compostos por múltiplos módulos comportamentais. A **Arquitetura de Subsunção**, proposta por Rodney Brooks, **organiza os comportamentos em camadas hierárquicas, nas quais camadas superiores podem suprimir ou inibir a atividade das inferiores**.

Por exemplo, num agente explorador, o comportamento *Regressar* pode suprimir *Vaguear* após a recolha de um alvo. Esta abordagem favorece o desenvolvimento incremental e confere robustez ao sistema, embora possa tornar-se complexa à medida que o número de comportamentos e interações entre camadas aumenta.

III. Agentes Reativos com Memória e Estado

Embora os agentes reativos puros não mantenham representações internas do mundo, a introdução de memória permite a emergência de comportamentos mais complexos, tais como evitar locais previamente explorados ou contornar situações problemáticas (como, por exemplo, veículos de Braitenberg presos em cantos). A memória é utilizada para registar perceções passadas e influenciar reações futuras, viabilizando:

1. **Comportamentos Temporais**: respostas que dependem não apenas da perceção atual, mas também do histórico de estímulos ou ações anteriormente executadas;
2. **Evitar Repetições**: tal como num aspirador robótico que cobre sistematicamente uma área, evitando revisitar zonas já limpas;
3. **Máquinas de Estados**: os comportamentos podem ser formalizados através de máquinas de estados finitos, onde as transições entre estados são determinadas por perceções e pela memória interna do agente

Contudo, a introdução de mecanismos de manutenção de estado implica um acréscimo de complexidade, tanto ao nível do espaço de memória como do processamento computacional. Ainda assim, esta abordagem não permite, por si só, a manipulação de representações simbólicas complexas nem o planeamento deliberado de ações alternativas. A evolução da Arquitetura de Subsunção para incorporar memória exemplifica esta transição, combinando a reatividade com mecanismos hierárquicos de controlo, como a inibição, a supressão e o reinício de comportamentos, proporcionando um maior grau de adaptabilidade e robustez comportamental.

Reação - Arquitetura

A **Reação - Arquitetura** nos agentes reativos é baseada na ligação direta entre perceção do ambiente e ação, sem recurso a planeamento ou representação interna complexa.

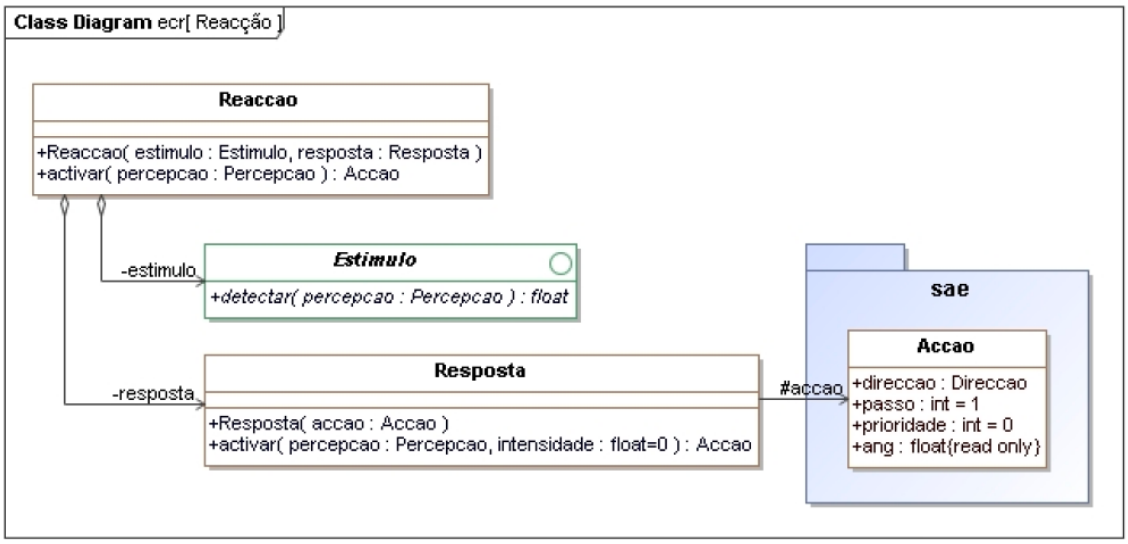


Figura 14 - Diagrama de classes | ecr[Reação]

As componentes principais desta arquitetura são:

- **Estimulo**: interface abstrata que define o método *detectar(percepcao)*. Cada subclasse implementa a lógica para detetar um estímulo específico (ex: obstáculo, alvo) a partir da perceção do ambiente;
- **Resposta**: classe que representa a ação a executar. O método *activar(percepcao, intensidade)* gera uma ação (por exemplo, mover, rodar), podendo ajustar a prioridade da ação conforme a intensidade do estímulo;
- **Reaccao**: liga um Estimulo a uma Resposta. Implementa o método activar(percepcao), que:
    - Usa o Estimulo para detetar a intensidade do estímulo na perceção;
    - Se a intensidade for maior que zero, ativa a Resposta, passando a perceção e a intensidade;
    - Retorna a ação gerada pela Resposta

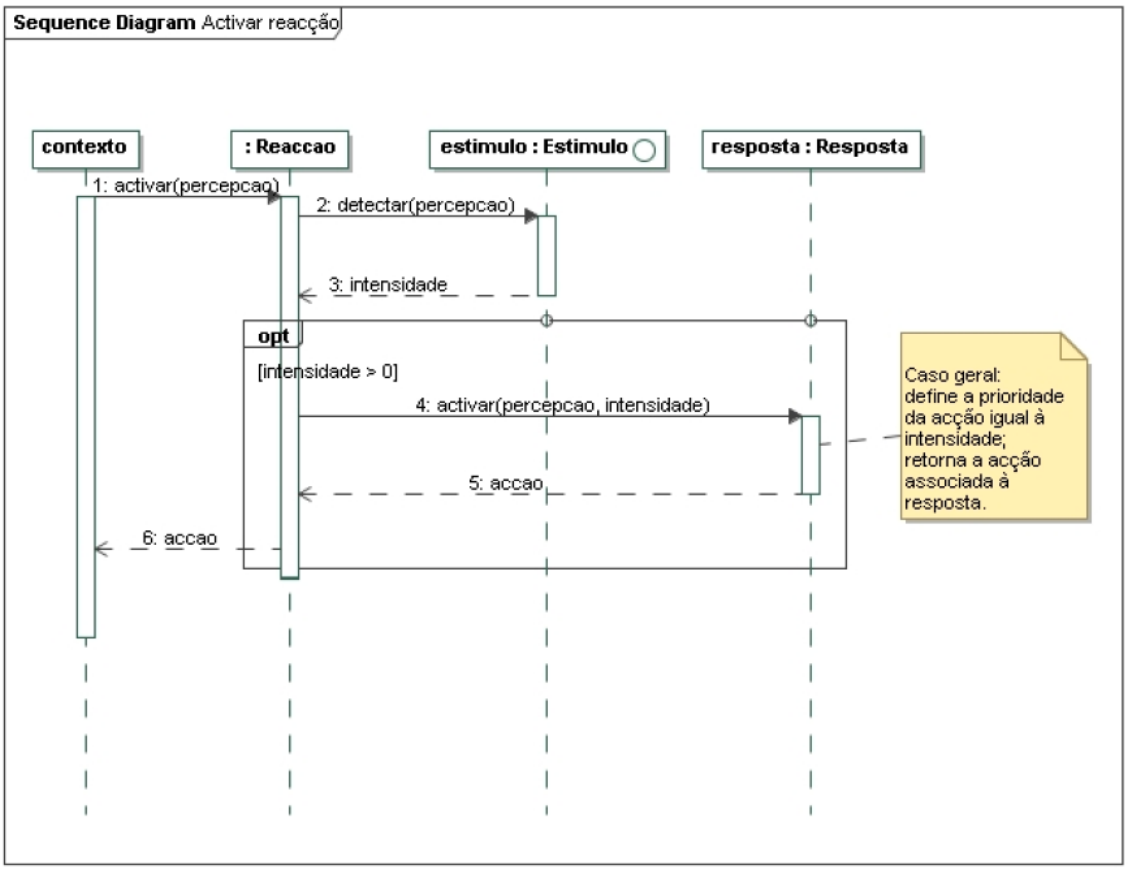


Figura 15 - Digrama sequencial de funcionamento do método activar(percepcao)

O diagrama indica que o método *activar(percepcao)* deteta se há algum alvo ou obstáculo, processando a intensidade da perceção. Caso a intensidade seja maior que 0, isto é, exista um alvo ou obstáculo, é criada uma ação em resposta a essa perceção e à sua intensidade.

Como já foi mencionado anteriormente, esta arquitetura de reação permite que o agente responda de forma rápida e robusta a mudanças no ambiente, através de módulos simples e independentes, alinhando-se com os princípios dos agentes reativos.

Esquemas Comportamentais Reativos

Os **Esquemas Comportamentais Reativos** são a base da arquitetura dos agentes reativos, definindo como o agente transforma perceções do ambiente em ações, de forma modular, rápida e sem planeamento deliberativo. O objetivo é criar agentes capazes de responder automaticamente a estímulos, através de associações diretas entre perceção e ação.

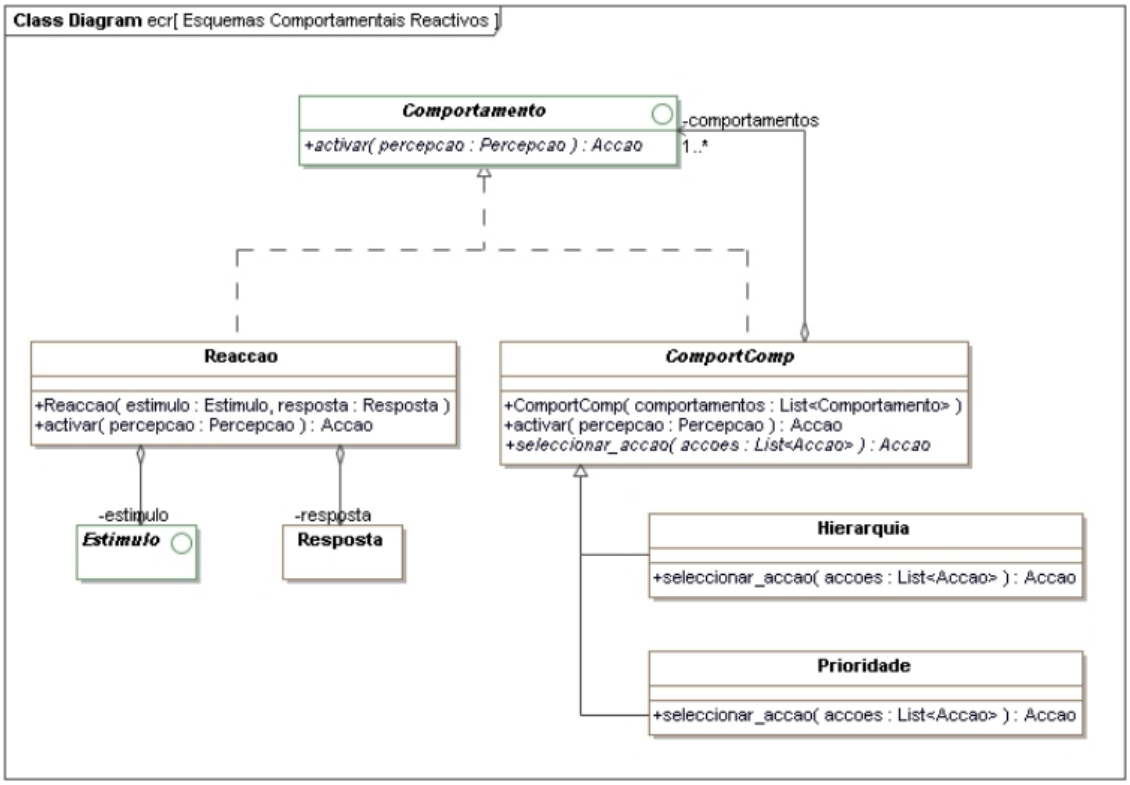


Figura 16 - Diagrama de classes | Esquemas Comportamentais Reativos

Este esquema é uma extensão da arquitetura de reação dos agentes reativos, incorporando:

- **Comportamento**: é uma interface abstrata para qualquer comportamento reativo. Um comportamento é definido como um conjunto de reações interligadas que, em resposta à perceção do ambiente, contribuem para a realização de uma ação específica;
- **Reaccao, Estimulo e Resposta** (esclarecidos previamente);
- **ComportComp:** é uma classe que representa um comportamento que agrega vários comportamentos, sejam eles simples ou compostos.

Para selecionar a ação final de entre as várias propostas, foram implementadas classes que definem uma estratégia de seleção, sendo elas:

- **Hierarquia:** como já foi explicado, esta classe é uma das estratégias de escolha de uma ação, onde as ações estão organizadas numa hierarquia fixa de subsunção, isto é, segue uma ordem fixa, escolhendo a primeira ação válida. Esta especializa de ***ComportComp***, e o seu método *seleccionar_accao*, retorna sempre a primeira ação da lista de ações passada como parâmetro;
- **Prioridade:** mencionada em "**Mecanismos de Reação e Coordenação de Comportamentos**", esta estratégia escolhe a ação com maior prioridade. É de esperar que esta classe especialize de ***ComportComp***, e que o seu método *seleccionar_accao* retorne sempre a ação com maior prioridade da lista de ações, usando a função ***max()*** e uma chave de avaliação, sendo ela o atributo "prioridade" associado a cada ação

Comportamento Reativo e Comportamento Explorar

O comportamento reativo é uma abordagem fundamental onde as ações do agente resultam de uma ligação direta e imediata entre perceções do ambiente e respostas pré-definidas, sem recurso a planeamento ou representação interna complexa. Este paradigma permite criar agentes robustos, rápidos e adaptáveis, capazes de responder automaticamente a estímulos do meio envolvente.

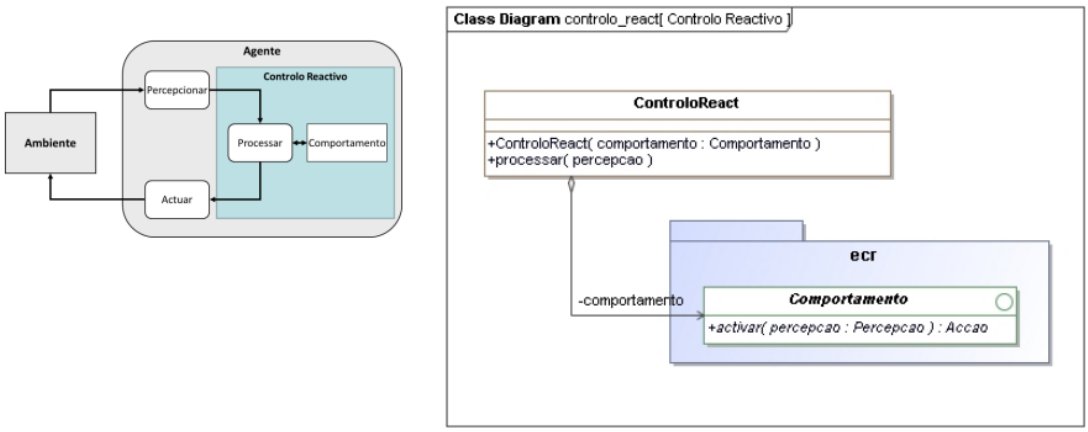


Figura 17 - Diagrama de classes | Controlo Reativo

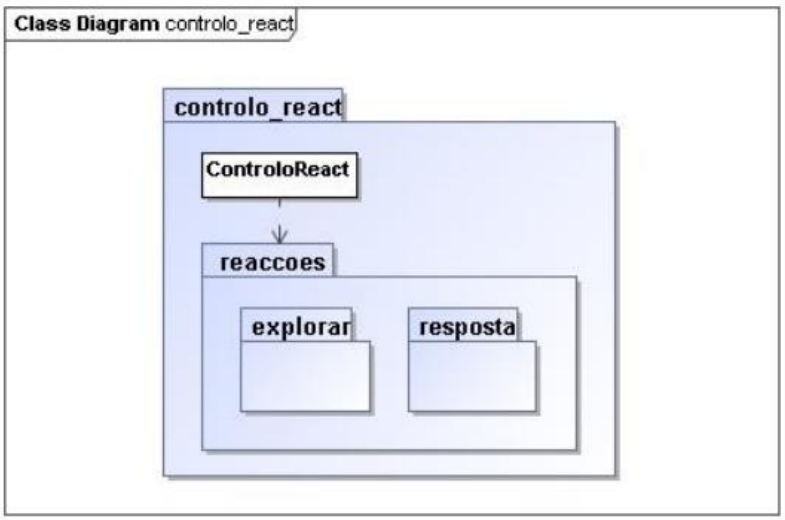


Figura 18 - Diagrama de classes do subsistema Controlo Reativo

A classe ***ControloReact*** representa o núcleo do agente reativo, onde o mesmo recebe perceções do ambiente e devolve ações, delegando a decisão do comportamento principal. Este necessita de uma instância da classe *Comportamento*, para guardar o comportamento a processar. Os métodos desta classe são:

- O **construtor**, onde recebe o comportamento a ser processado, seja ele simples ou composto;
- ***processar(percepcao)***, onde o mesmo chama a função ***activar(percepcao)*** no comportamento, devolvendo a ação a executar

Dentro deste contexto, destaca-se o comportamento ***Explorar***, responsável por permitir ao agente movimentar-se de forma aleatória ou sistemática quando não existem estímulos prioritários (como obstáculos ou alvos). Este comportamento é essencial para garantir que o agente cobre novas áreas do ambiente, aumentando a sua eficácia em tarefas de procura ou recolha.

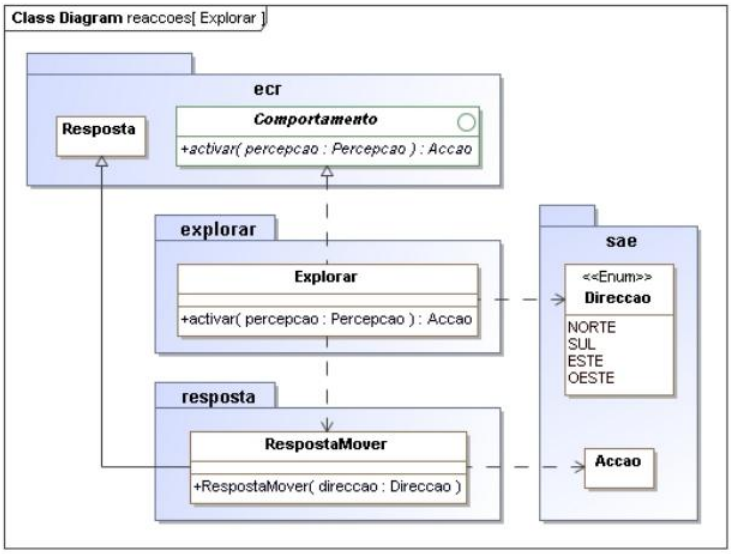


Figura 19 - Diagrama de classes | Explorar

A classe *Explorar* especializa de *Comportamento*, e foi implementada de modo a que o agente se possa mover aleatoriamente pelo ambiente. Este movimento aleatório é possível graças ao método *activar(percepcao)* que, mesmo não usando a perceção, pois não é necessária, ao gerar um número aleatório entre 0 e 1, se esse valor por maior que a probabilidade de rotação passada como parâmetro no construtor, o agente roda numa direção aleatória que esteja presente na lista de direções (NORTE, SUL, ESTE, OESTE). Caso seja maior ou igual, o agente apenas avança em frente.

Neste diagrama também está presente a classe ***RespostaMover***, que herda de *Resposta*. Esta representa uma resposta reativa do agente que consiste em mover-se numa direção específica. É utilizada para gerar ações de movimento em comportamentos reativos. O único método presente é o próprio construtor, onde, ao usar a super classe, cria uma ação (Accao()) com a direção indicada e passo igual 1, garantindo que o movimento ocorra a cada ativação.

O controlo reativo é finalmente estendido para receber mais reações, sendo elas:

- ***AproximarAlvo***: comportamento que utiliza um conjunto de reações direcionais baseadas nas direções previamente definidas na biblioteca SAE. Este comportamento utiliza também ***AproximarDir*** (que irá ser falado mais à frente), que faz com que o agente se aproxime numa determinada direção;
- ***EvitarObst***: representa uma reação simples para evitar obstáculos. Este utiliza ***EvitarDir*** (será também abordado posteriormente) que faz com que um agente evite um obstáculo numa certa direção

Por fim, foi implementada a classe ***Recolher***, que é um comportamento composto que organiza os comportamentos internos pela sua prioridade.

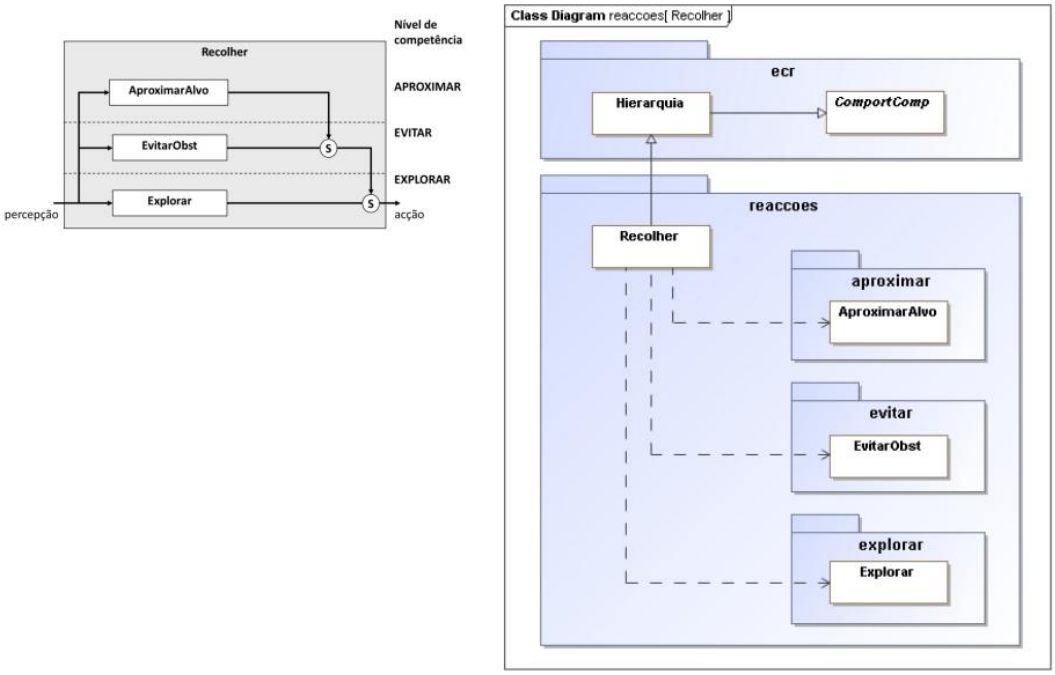


Figura 20 - Diagrama de classes | Recolher

Comportamentos Compostos

Como já foi mencionado anteriormente, comportamentos compostos são estruturas fundamentais em agentes reativos que permitem combinar e coordenar múltiplos comportamentos simples (reações) para gerar decisões mais robustas e adaptativas. Em vez de depender de uma única regra estímulo-resposta, o agente pode integrar várias reações (como evitar obstáculos, aproximar-se de alvos ou explorar) e decidir qual executar com base em mecanismos como **prioridade** ou **hierarquia**. Isto permite modularidade, escalabilidade e maior flexibilidade no controlo do agente.

Nesta parte, foram implementadas as *packages*:

- *aproximar*, que contém as reações:
    - *AproximarAlvo* (falada anteriormente);
    - *AproximarDir* – reação que faz com que o agente se aproxime de algo numa direção específica. Esta reação utiliza um estímulo para detetar alvos e uma resposta para mover o agente na direção escolhida;
- *evitar*, que inclui as reações:
    - *EvitarObst* (falada anteriormente);
    - *EvitarDir* – reação que permite ao agente evitar um obstáculo numa direção particular. Esta usa um estímulo para detetar um obstáculo no ambiente e uma resposta para o agente evitar o devido obstáculo na direção selecionada

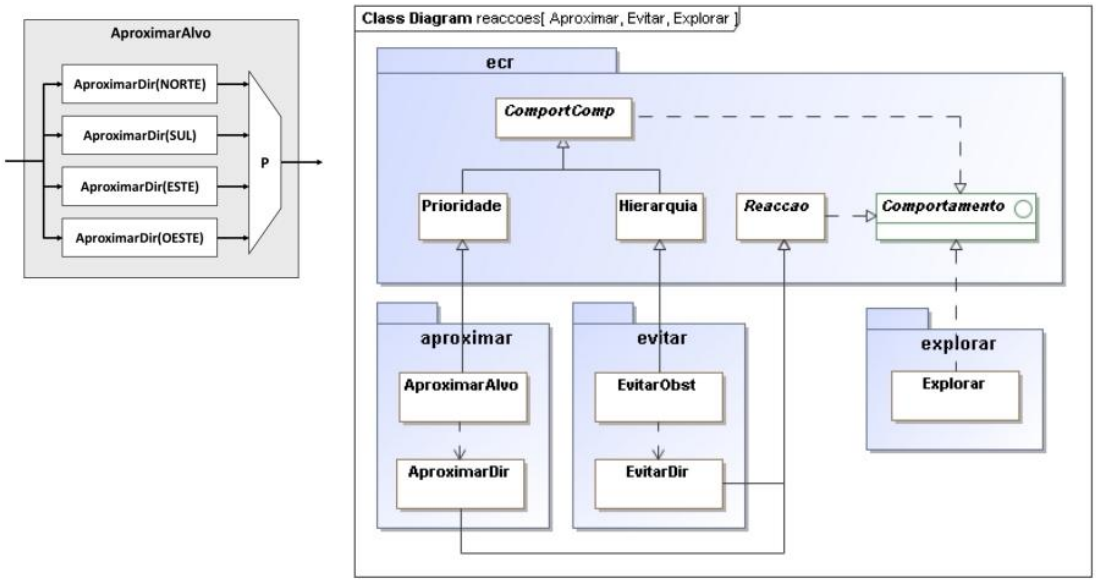


Figura 21 - Diagrama de organização das reações

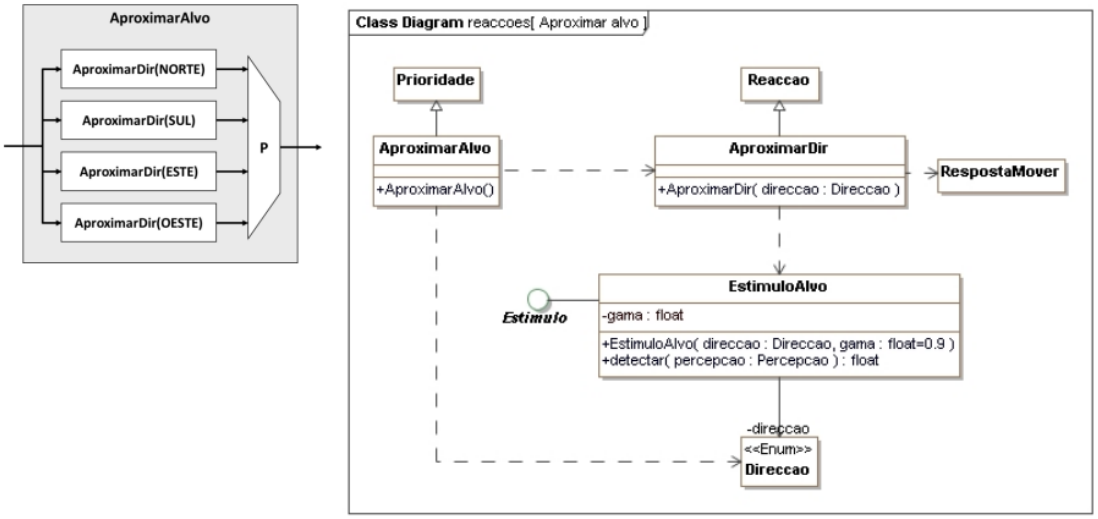


Figura 22 - Diagrama de classes | Aproximar Alvo

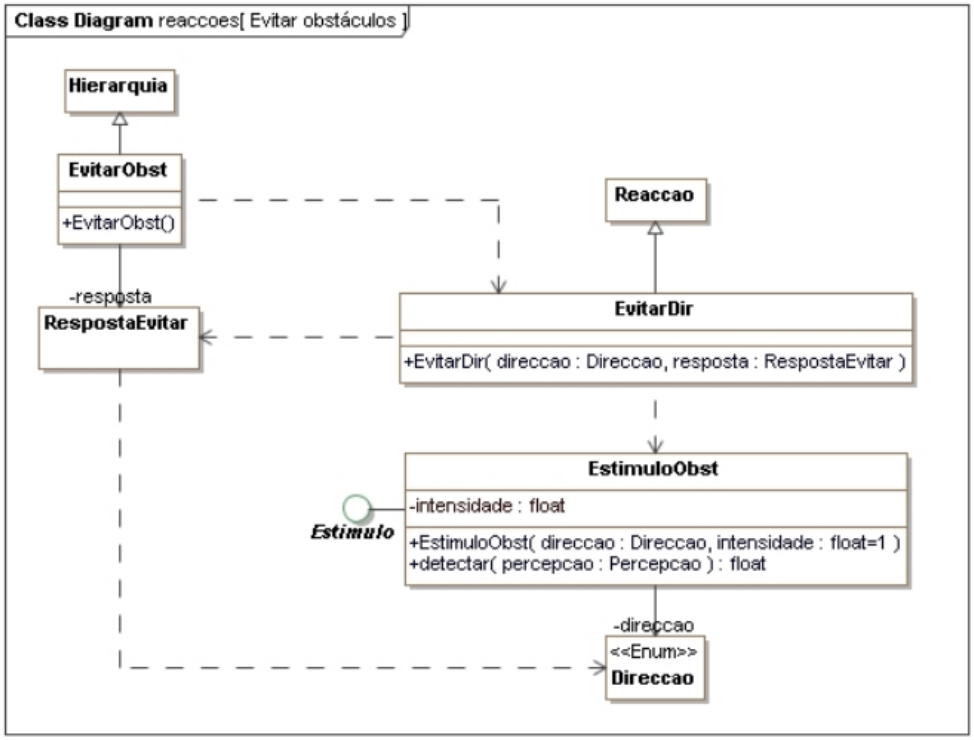


Figura 23 - Diagrama de classes | Evitar Obstáculos

Para o agente poder aproximar de um alvo e evitar obstáculos, foi necessário, também, implementar estímulos para cada reação. Esses estímulos foram:

- ***EstimuloAlvo*** – é uma classe que deteta a presença de um alvo numa direção específica e que calcula a intensidade desse estímulo em função da distância ao alvo. Esta classe recebe, no construtor, uma direção de movimento e um fator de decaimento (valor que serve para reduzir a intensidade do estímulo à medida que a distância ao alvo aumenta). O método *detectar(percepcao)* lê da

perceção o elemento do ambiente, a distância e ignora o resto para a direção configurada. Caso o elemento seja um alvo, é retornado como valor da intensidade $gama^{distância}$, isto é, a intensidade decresce com a distância até ao alvo, caso contrário, a intensidade é 0;

- ***EstimuloObst*** – é uma classe que deteta a presença de um obstáculo numa direção específica e devolve uma intensidade fixa se houver obstáculo. Esta classe recebe, no construtor, a direção a controlar e a intensidade (valor a devolver se houver obstáculos, que por defeito é 1). Já o método *detectar(percepcao)* percebe se o agente está a colidir com um obstáculo, e caso esteja, devolve a intensidade passada como parâmetro. Se não existir obstáculo, retorna 0

Respostas a estímulos

Como já foi resumidamente mencionado, para os sub-comportamentos de aproximar e evitar originarem uma ação, foi necessário definir uma resposta a cada estímulo, onde:

- ***RespostaMover*** – classe que especializa de *Resposta* e que representa a resposta do agente ao estímulo de se mover numa determinada direção. Esta recebe a direção para onde o agente se deve mover e cria uma ação com passo igual a 1, garantindo que o movimento ocorra a cada ativação. Ao chamar o construtor da super classe, *RespostaMover* ativa a ação de movimento utilizando o método *activar(percepcao, intensidade=0)*;
- ***RespostaEvitar*** – também especializa de *Resposta*. Esta classe mostra a resposta do agente ao detetar um obstáculo: rodar para uma nova direção de modo a evitar colisão. O método *activar(percepcao)* funciona da seguinte maneira:
    1. Obtém a direção atual do agente;
    2. Verifica se há obstáculo nessa direção;
    3. Se existir, calcula uma nova direção, de modo a evitar o obstáculo;
    4. Retorna a ação de rodar para evitar o obstáculo

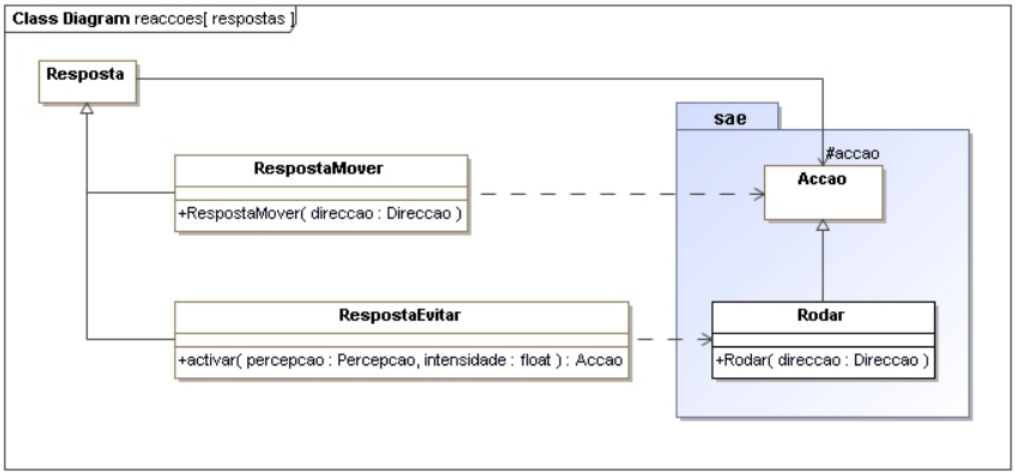


Figura 24 - Diagrama de classes das respostas

Exploração com memória

A exploração com memória é uma extensão dos comportamentos reativos simples, permitindo ao agente registar estados anteriormente visitados. O objetivo deste comportamento, para além do que já foi mencionado, é tornar a exploração do ambiente mais eficiente e menos redundante. Devido à memória ser limitada, o agente só se consegue lembrar de um número finito de situações passadas, o que faz com que a complexidade aumente, tanto a nível de memória como de processamento computacional.

Para implementar este comportamento, criou-se a classe ***ExplorarComMem***, onde:

- O construtor recebe um valor para definir a memória máxima do agente (que por defeito é 100), inicia uma lista para armazenar as situações já visitadas pelo agente (memória), e define a resposta padrão como avançar;
- Já no método *activar(percepcao)*, são extraídas a posição e direção atuais do agente, ou seja, a sua situação atual no ambiente;
- Caso essa situação não esteja na memória do agente, a mesma é adicionada, e se a memória exceder o limite, a situação mais antiga é removida;
- Finalmente, é ativada a resposta padrão definida no construtor

Agente Reativo Implementado

Para finalmente testar estas implementações, implementou-se um agente reativo tendo em conta o seu ciclo fundamental: percecionar -> processar -> atuar, isto é, obter informações do ambiente, transformar perceções em ações, e executar a ação no ambiente.

A classe ***AgenteReact*** especializa de ***Agente*** (presente na biblioteca SAE). Esta usa um módulo de controlo reativo para transformar perceções do ambiente em ações. O funcionamento da mesma é:

- O construtor inicializa o agente, chamando o construtor da classe *Agente*; cria o comportamento principal, o *Recolher*, que incluí todos os sub-comportamentos, organizados hierarquicamente:
    1. AproximarAlvo <- prioridade máxima;
    2. EvitarObst <- prioridade média;
    3. ExplorarComMem <- prioridade baixa;
    4. Explorar (com probabilidade de rotação igual a 0.7) <- prioridade mínima;

  E ainda inicia o módulo de controlo reativo (citado previamente), onde é passado como parâmetro o comportamento do agente, ou seja, o *Recolher*;
- Já no método ***executar()*** (método que implementa o ciclo percecionar -> processar -> atuar), é obtida a perceção do ambiente (usando o método ***_percecionar()*** herdado de *Agente*), processa-se essa perceção (método

processar(percepcao) de *ControloReact*) de modo a gerar uma ação, e por fim, utilizou-se o método herdado ***_actuar(accao)*** para executar a ação no ambiente

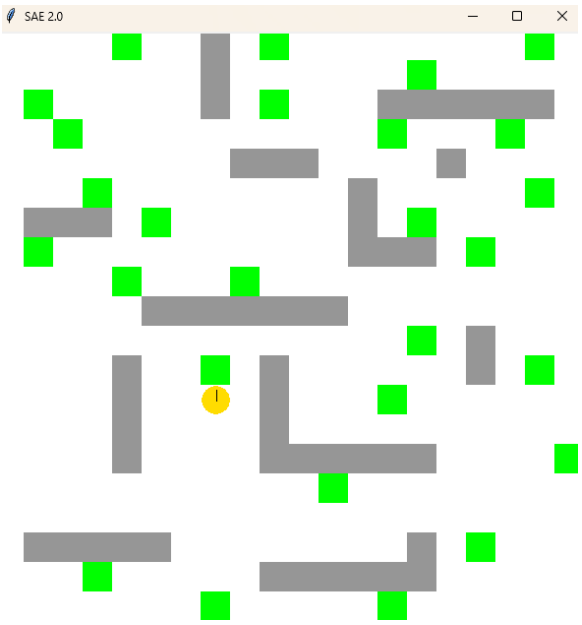


Figura 25 - Execução do Agente Reativo

Parte 3

A Parte 3 do projeto tem como objetivo principal a **implementação do raciocínio automático com base em PEE (Procura em Espaço de Estados)**. Esta abordagem tem como objetivo dotar o agente de capacidade para encontrar soluções através da exploração sistemática de estados possíveis.

É introduzido o **modelo de problema**, onde são definidos atributos como a identificação única dos estados e restrições de acesso à informação. A arquitetura inclui o **subsistema PEE**, que integra mecanismos fundamentais de procura, como a **inicialização da memória**, **gestão da fronteira de exploração**, **memorizar e comparar nós**, e ainda diferentes estratégias de procura: **procura informada e não informada.**

Além disso, são considerados os **avaliadores de nós**, componentes que determinam a prioridade na exploração de caminhos consoante o tipo de procura utilizado. Esta parte é crucial para a construção de agentes deliberativos capazes de planear sequências de ações de forma eficiente, baseada em critérios definidos.

A implementação visa resolver problemas de planeamento de forma eficiente, considerando otimização e complexidade computacional, conforme os princípios discutidos nos documentos de suporte.

Raciocínio Automático

I. Introdução ao Raciocínio Automático

O raciocínio automático constitui uma área da inteligência artificial dedicada à resolução de problemas através de métodos algorítmicos, com recurso a representações simbólicas do conhecimento e a mecanismos formais de inferência. Esta abordagem permite que sistemas computacionais simulem capacidades de deliberação e tomada de decisão, sendo aplicada em múltiplos domínios, como o planeamento robótico, a logística, o controlo de personagens em ambientes virtuais e a resolução de problemas matemáticos.

O processo de raciocínio automático envolve a exploração e avaliação sistemática de diferentes alternativas, com o objetivo de identificar uma solução válida ou ótima. Para tal, o sistema recorre a modelos internos que representam, de forma estruturada, o domínio do problema, permitindo inferir consequências, avaliar hipóteses e selecionar cursos de ação com base em critérios lógicos e/ou heurísticos.

II. Componentes Fundamentais

O raciocínio automático fundamenta-se em três conceitos centrais que estruturam a resolução de problemas de forma sistemática:

- **Estado**: representa uma configuração única do problema num dado momento. Por exemplo, a disposição das peças num puzzle ou a posição de um robô num ambiente;
- **Operador**: define uma ação ou transformação que, aplicada a um estado, conduz a outro estado. Um exemplo seria o movimento de uma peça num jogo de blocos;
- **Espaço de Estados**: corresponde ao conjunto de todos os estados possíveis e às transições entre eles, geralmente representado sob a forma de um grafo, onde os nós correspondem aos estados e as arestas às operações permitidas

A formalização de um problema de raciocínio automático requer os seguintes elementos:

- Um **estado inicial**, que define a configuração a partir da qual o sistema inicia o processo de resolução;
- Um **conjunto de operadores**, que especifica as ações permitidas para transformar estados;
- Um **objetivo**, que pode ser expresso como um estado final desejado ou como uma função objetivo, capaz de avaliar e comparar a qualidade ou adequação de diferentes estados no espaço de soluções

III. Processo de Resolução de Problemas

A resolução automática de problemas envolve, de forma geral, duas etapas fundamentais: a **exploração de opções** e a **avaliação de opções**.

- **Exploração de Opções**
  - Nesta fase, o sistema realiza um raciocínio prospetivo, simulando mentalmente as possíveis consequências de cada ação. Esta capacidade de antecipação exige a existência de uma representação interna do domínio, que permita prever os estados resultantes da aplicação dos operadores;
- **Avaliação de Opções**
  - Após a geração de alternativas, o sistema procede à avaliação de cada uma com base em dois critérios principais:
    - **Custo**: refere-se aos recursos necessários para executar uma determinada ação ou sequência de ações, podendo incluir tempo de execução, consumo de energia, ou complexidade computacional;
    - **Valor**: representa o benefício ou utilidade associada à obtenção de um determinado estado, muitas vezes expresso em termos de proximidade ao objetivo ou qualidade da solução

IV. Representação do Conhecimento

Para que o raciocínio automático seja viável, é fundamental converter a informação concreta do problema em estruturas simbólicas internas, adequadas à manipulação algorítmica. Este processo envolve duas fases principais:

- **Codificação**: consiste na tradução do problema do mundo real para uma representação simbólica manipulável pelo sistema. Por exemplo, a configuração de um puzzle pode ser representada através de uma matriz, facilitando a aplicação de operadores e a exploração do espaço de estados;
- **Descodificação**: corresponde à conversão da solução simbólica obtida pelo sistema em ações concretas no domínio físico ou aplicado. Um exemplo seria a transformação de uma sequência de movimentos calculados simbolicamente em comandos motores para um braço robótico

A qualidade da representação simbólica é crucial para a eficácia do raciocínio automático, devendo capturar de forma explícita tanto a **estrutura** do problema — isto é, os estados possíveis — como a sua **dinâmica**, expressa através dos operadores de transição entre estados. Uma representação adequada permite ao sistema inferir, planear e agir de forma coerente e eficiente face aos objetivos definidos.

Procura em Espaços de Estados

A procura em espaços de estados é um método fundamental na resolução de problemas de planeamento, onde o principal objetivo é determinar sequências de ações que permitam transitar de um estado inicial para um estado final.

I. Problemas de Planeamento

Os problemas de planeamento consistem em encontrar uma sequência de ações que permita transitar de um estado inicial para um estado objetivo. Estes problemas são frequentemente representados através de uma abstração do domínio como um espaço de estados, no qual cada estado representa uma configuração possível do sistema e as ações correspondem a operadores que geram transições entre estados. A solução para este tipo de problema traduz-se, assim, num percurso dentro desse espaço, ligando o estado inicial ao estado objetivo através de sucessivas aplicações de operadores.

II. Processo de Procura

A resolução de problemas de planeamento envolve a exploração sistemática do espaço de estados, representado como um grafo em que os vértices correspondem a estados e os arcos representam transições entre eles. O processo inicia-se na verificação de se o estado atual corresponde ao objetivo. Caso não corresponda, procede-se à expansão do estado atual, gerando todos os estados sucessores possíveis. Este procedimento repete-se para cada novo estado gerado até que se atinja o objetivo ou até que se esgotem todos os estados disponíveis para exploração, indicando assim a inexistência de uma solução.

III. Árvore de Procura

A árvore de procura é uma estrutura que organiza a informação gerada durante o processo de procura, representando-a sob a forma de nós. Cada nó contém dados relativos ao estado associado, ao operador que originou esse estado, ao nó antecessor, à profundidade a que se encontra e ao custo acumulado até ao momento. Esta estrutura tem como principal função manter a coerência do processo de procura, estabelecendo ligações entre nós sucessivos e armazenando a informação necessária para identificar o percurso seguido até à solução.

IV. Fronteira de Exploração

A fronteira de exploração é uma estrutura de dados fundamental nos algoritmos de procura em espaços de estados. Esta estrutura mantém os nós que já foram gerados, mas que ainda não foram expandidos, sendo responsável por determinar a ordem pela qual o algoritmo prossegue com a exploração do espaço de estados.

V. Métodos de Procura Não Informada

Os mecanismos de procura não informada são algoritmos que exploram o espaço de estados sem recorrer a qualquer conhecimento adicional sobre a localização do objetivo, baseando-se apenas na descrição formal do problema.

1. **Procura em Profundidade (*Depth-First Search* – DFS):** a procura em profundidade segue uma estratégia de exploração que dá prioridade aos nós mais recentes, avançando o mais profundamente possível num ramo antes de recuar. Utiliza uma fronteira do tipo LIFO (*Last-In*, *First-Out*), ou seja, uma pilha, onde os nós sucessores são inseridos no início da estrutura. Embora seja simples de implementar e exija pouca memória, esta abordagem não é completa em espaços infinitos ou com ciclos, pois pode entrar em *loops*. Também não garante uma solução ótima, já que pode encontrar uma solução mais profunda antes de uma mais curta. É útil em problemas onde a profundidade máxima é limitada ou quando não se exige ser ótimo;
2. **Procura em Largura (*Breadth-First Search* - BFS):** a procura em largura explora todos os nós de um dado nível antes de avançar para o nível seguinte, utilizando uma fronteira FIFO (*First-In*, *First-Out*), ou seja, uma fila. Cada vez que se expandem nós, os sucessores são adicionados ao final da fila. Esta abordagem é completa — garante encontrar uma solução se ela existir — e também ótima quando os custos das ações são todos iguais. No entanto, tem uma complexidade temporal e espacial elevada, o que a torna ineficiente para espaços de estados muito grandes. Ainda assim, é indicada quando se acredita que a solução se encontra perto do estado inicial;
3. **Procura em Profundidade Limitada (*Depth-Limited Search* - DLS)**: a DLS é uma variação da DFS em que se impõe um limite l à profundidade máxima de exploração. Isso evita que o algoritmo entre em ciclos ou se perca em ramos infinitos. Tal como na DFS, usa uma pilha (LIFO) e insere os sucessores no topo da fronteira. A procura é completa apenas se a solução estiver dentro do limite definido, e continua a não ser ótima;
4. **Procura em Profundidade Iterativa (*Iterative Deepening Search* - IDS)**: a IDS combina as vantagens da BFS e da DFS ao executar múltiplas procuras em profundidade limitada com limites crescentes até encontrar a solução. Esta abordagem é completa e ótima (para custos uniformes). Apesar de repetir o trabalho em iterações anteriores, a sobrecarga é geralmente pequena, tornando-a ideal para problemas em que a profundidade da solução é desconhecida;
5. **Procura de Custo Uniforme (*Uniform-Cost Search* - UCS)** a UCS expande sempre o nó com o menor custo acumulado desde o estado inicial, garantindo assim que o caminho mais barato é explorado primeiro. Utiliza uma fronteira ordenada pelo custo total do caminho, diferindo da BFS ao considerar custos diferentes para as ações. Esta abordagem é completa e

ótima, independentemente da profundidade da solução, desde que todos os custos sejam positivos;

Método de Procura	Tempo	Espaço	Ótimo	Completo
Profundidade	$O(b^m)$	$O(bm)$	Não	Não
Largura	$O(b^d)$	$O(b^d)$	Sim	Sim
Custo Uniforme	$O(b^{[C*/\varepsilon]})$	$O(b^{[C*/\varepsilon]})$	Sim	Sim
Profundidade Limitada	$O(b^l)$	$O(bl)$	Não	Não
Profundidade Iterativa	$O(b^d)$	$O(bd)$	Sim	Sim

Tabela 1 - Complexidade Computacional dos métodos de procura não informada

$b - fator\ de\ ramificação$ $d - dimensão\ da\ solução$ $m - profundidade\ da\ árvore\ de\ procura$	$C^* - Custo\ da\ solução\ ótima$ $\varepsilon - Custo\ mínimo\ de\ uma\ transição\ de\ estado$

VI. Métodos de Procura Informada e Função Heurística

Nos domínios do planeamento automático e da resolução de problemas complexos, os métodos de procura são fundamentais para encontrar caminhos entre um estado inicial e um estado objetivo. Ao contrário dos métodos de procura não informada, que exploram o espaço de estados sem qualquer orientação, os métodos de **procura informada** utilizam conhecimento adicional do problema para guiar a pesquisa de forma mais eficiente. Este conhecimento é incorporado através de *funções heurísticas*, que permitem reduzir significativamente o número de estados explorados e, consequentemente, o tempo de execução.

As funções heurísticas $h(n)$ são elementos centrais na procura informada, pois estimam o custo até ao objetivo a partir de um dado estado n. Com base nessas estimativas, os algoritmos conseguem priorizar os caminhos mais promissores. Para garantir a eficácia da procura, a heurística deve ser *admissível* (nunca sobrestima o custo real) e idealmente *consistente* (satisfaz $h(n) \leq c(n, n') + h(n')$, onde n' é o nó sucessor de n, e $c(n, n')$ é o custo de transição entre os estados). Estas propriedades asseguram, respetivamente, que a solução encontrada tem o menor custo possível (isto é, é ótima) e que a procura é eficiente ao evitar repetições desnecessárias.

Que tipos de procura informada existem?

1. **Procura Melhor-Primeiro (*Best-First Search*)**: a procura melhor-primeiro é uma estratégia geral que expande, a cada iteração, o nó considerado mais promissor segundo uma função de avaliação $f(n)$. Esta função pode ser definida de diversas formas, dependendo do tipo de conhecimento disponível sobre o problema. Por exemplo, pode representar o custo acumulado até ao nó ($f(n) =$

$g(n)$), a estimativa heurística até ao objetivo ($f(n) = h(n)$), ou a combinação de ambos ($f(n) = g(n) + h(n)$). A fronteira de exploração é organizada como uma fila de prioridade, onde os nós com menor valor de $f(n)$ são explorados primeiro. Esta abordagem fornece a base para variantes como a Procura de Custo Uniforme, a Procura Sôfrega e o algoritmo A*, sendo, por isso, fundamental no contexto da procura informada;

2. **Procura Sôfrega (*Greedy Search*)**: a procura sôfrega utiliza exclusivamente a heurística $h(n)$ como função de avaliação, ou seja, $f(n) = h(n)$, orientando a exploração dos estados com base na estimativa do custo restante até ao objetivo. Esta abordagem pode ser bastante eficiente em termos computacionais, especialmente quando a heurística é bem definida, mas não é completa (pode falhar em espaços com ciclos ou infinitos) nem garante a obtenção de soluções ótimas. Por priorizar caminhos que *parecem* promissores, pode ignorar alternativas com menor custo total. Assim, é mais indicada para problemas onde a rapidez é mais importante do que a qualidade da solução;
3. **Procura A***: o algoritmo A* combina as vantagens da procura informada e da procura de custo acumulado, utilizando a função $f(n) = g(n) + h(n)$, onde $g(n)$, representa o custo real até ao nó n, e $h(n)$ é a estimativa heurística até ao objetivo. Se a heurística for *admissível* — isto é, nunca sobrestima o custo real —, A* garante que a solução encontrada é ótima. Se for *consistente* (ou monótona), ou seja, satisfaz $h(n) \leq c(n, n') + h(n')$ para todos os nós n e seus sucessores n', então o algoritmo evita reexplorações desnecessárias. A* é completo e encontra sempre a solução de menor custo, sendo um dos algoritmos mais eficientes entre aqueles que asseguram a obtenção da melhor solução possível. Contudo, pode apresentar complexidade exponencial no pior caso, embora este impacto seja significativamente reduzido quando se recorre a heurísticas bem concebidas

Implementação

Modelo de Problema

O **modelo de problema** é uma abstração fundamental na área de inteligência artificial para a **resolução automática de problemas através de algoritmos de procura em espaço de estados**. O modelo define os principais elementos necessários para descrever qualquer problema de procura: **o estado inicial, o conjunto de operadores (ações possíveis) e objetivo**.

Primeiramente, realizou-se a classe abstrata ***Problema*** que define a estrutura base para qualquer problema a ser resolvido pelos algoritmos anteriormente mencionados. Esta encapsula os elementos essenciais para a descrição de um problema: o estado inicial, os operadores e o objetivo.

A classe funciona da seguinte maneira:

- Construtor:
    - Recebe como parâmetros o **estado inicial** do problema (sendo este uma instância da classe ***Estado***, que irá ser falada posteriormente), e a lista de operadores (instância da classe ***Operador***, que também irá ser falada depois) que definem as ações possíveis, armazenando as mesmas em atributos privados. É também feita uma verificação para perceber se existe pelo menos um operador;
- Método abstrato ***objectivo(estado)***:
    - Este método, sendo abstrato, é implementado consoante a estratégia de procura, e serve para perceber se um determinado estado é objetivo, de modo a indicar se se deve parar a procurar;
- Propriedades ***estado_inicial*** e ***operadores***:
    - Estas propriedades permitem o acesso apenas para leitura aos atributos privados, garantindo o encapsulamento e integridade dos dados;

De seguida, implementou-se a classe abstrata ***Estado*** que representa a estrutura base de qualquer configuração possível num problema de procura. Cada instância desta classe corresponde a uma situação concreta do agente no mundo. No contexto de procura em espaço de estados, um estado corresponde a um nó da árvore de procura e é um dos vértices do grafo de procura. É fundamental distinguir estados diferentes e evitar repetições, por isso é necessário que cada estado tenha uma identificação única.

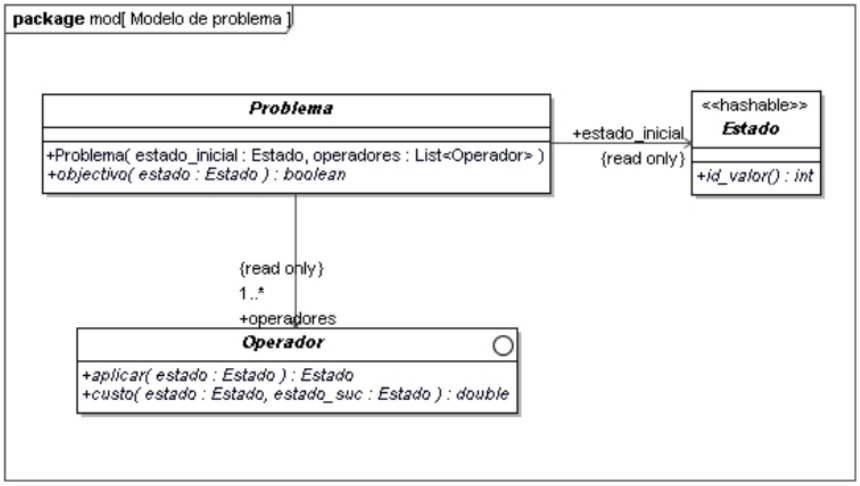
A classe contém os seguintes métodos:

- Método abstrato ***id_valor(self)***:
    - Como é abstrato, este método irá devolver um valor único que identifique inequivocamente cada configuração do problema;
- Método ***__hash__(self)***:
    - Este método retorna o valor do método anterior, e permite ainda o armazenamento eficiente em dicionários e conjuntos, e a deteção de estados repetidos durante a procura
- Método ***__eq(self, other)***:
    - Define a igualdade entre dois estados: dois estados são iguais se os seus identificadores forem iguais;

Por fim, implementou-se a interface ***Operador***, que define o contrato para todas as ações possíveis num problema de procura. Cada operador representa uma ação que pode ser aplicada a um estado, gerando um novo estado, e um custo associado a essa transição.

A interface realizada contém:

- O método ***aplicar(estado)***, que recebe um estado e retorna o novo estado resultando da aplicação do operador. Corresponde à definição de operadores como funções de transição de estados;
- O método ***custo(estado, estado_suc)***, que calcula e retorna o custo da transição entre o estado original e o estado sucessor;



{read only} – Restrição que indica acesso a atributo apenas para leitura (*propriedade de leitura*)
<<hashable>> – Estereótipo que indica a definição de mecanismos de identificação única
id_valor(): int – Define identificação única do estado em função da sua informação (*valor de estado*)

Figura 26 - Diagrama de classes do modelo de problema

Mecanismo de Procura

Um **mecanismo de procura** é o processo sistemático de explorar um espaço de estados para encontrar uma sequência de ações (ou operadores), desde um estado inicial até a um estado objetivo. Este mecanismo utiliza uma fronteira para gerir os nós a expandir e opera sobre um modelo de problema.

A classe ***MecanismoProcura*** é uma classe abstrata que define o algoritmo genérico de procura em espaços de estados. Os métodos aqui presentes:

- Construtor:
  - Inicializa o mecanismo com uma estratégia de fronteira, sendo que esta determina a estratégia de procura, ou seja, a ordem de exploração dos nós. O construtor recebe uma fronteira como parâmetro e armazena-a num atributo protegido;
- ***_iniciar_memoria()***:
  - Inicializa a fronteira, garantindo que a procura começa "do zero" a cada execução. O método chama a função iniciar pertencente à fronteira;
- ***_memorizar()***:
  - Regista um nó na estrutura de memória, permitindo controlar os nós abertos e fechados, evitando ciclos e repetições. Aqui é utilizado o método ***inserir(no)*** para inserir um nó na fronteira;
- ***procurar(problema)***:
  - Implementa o algoritmo geral de procura, onde é explorado o espaço de estados até encontrar uma solução. O método funciona da seguinte forma:
    - Inicialização da memória, isto é, inicia a fronteira;

        - Criação do nó inicial;
        - Memorizar o nó criado;
        - Enquanto existirem nós na fronteira:
            - Remove-se o primeiro nó da fronteira;
            - Verifica-se se o estado é objetivo;
            - Se for, retorna a solução;
            - Senão, expande o nó e adiciona sucessores à fronteira;
        - Retorna *None* se a fronteira esvaziar sem solução;
- ***_expandir(problema, no)***:
    - Gera nós sucessores aplicando operadores ao estado atual. Este foi implementado como:
        - Para cada operador do problema:
            - Aplicar o operador ao estado, gerando um sucessor;
            - Se gerar um estado válido, calcula-se o custo acumulado (custo do nó pai + custo da transição) e cria um nó com o estado sucessor, o operador, o nó antecessor e o custo acumulado;
        - Finalmente, retorna todos os sucessores gerados

Foram criadas, também, propriedades para aceder apenas por leitura aos nós que foram processados, aos nós em memória e aos nós repetidos.

Para além da classe *MecanismoProcura*, foi implementada também a classe abstrata ***Fronteira***. Esta define a estrutura de dados que mantém os nós a expandir. *Fronteira* conta com as seguintes funções:

- Construtor: inicializa a estrutura de dados da fronteira, chamando o método ***iniciar()***;
- ***iniciar()***: este método inicia a fronteira, definindo uma lista vazia para guardar os nós da fronteira;
- ***inserir(no)***: serve para inserir um nó na fronteira. Sendo este abstrato, a implementação vai depender da estratégia;
- ***remover()***: remove e retorna o próximo nó a expandir, utilizando a função *pop()*, que remove e retorna um item no índice indicado;

Foi definida também uma propriedade para perceber se a fronteira está vazia, ou seja, se não existem nós.

De seguida, foi criada a classe ***No***, que representa um nó na árvore de procura. Um nó é uma estrutura fundamental em algoritmos de procura que:

1. Representa uma configuração específica do problema (estado);
2. Armazena informações sobre como esse estado foi alcançado:
    a. Operador aplicado;
    b. Nó antecessor
3. Mantém métricas de procura:

a. Profundidade na árvore;
b. Custo acumulado do caminho;

Esta classe contém então:

- Construtor: onde são armazenados em atributos privados, as informações passadas como parâmetro, como o estado atual do problema, o operador que levou a esse estado, o antecessor e o custo acumulado do percurso. Também foi criado um atributo, também privado, que define a prioridade de um nó;
- Propriedades: para aceder apenas por leitura aos atributos criados no construtor;
- Devido à existência de fronteiras de prioridade, criou-se um *setter* para alterar o valor da prioridade;
- Para o avaliador poder comparar nós, implementaram-se os métodos ***__lt__(outro_no)***, que compara a prioridade entre nós, e foi também criado o método ***__del__()***, que acrescenta um nó à lista de nós eliminados

Por fim, foram produzidas duas classes: ***Solucao*** e ***PassoSolucao***.

A classe *Solucao*, como o próprio nome indica, representa o caminho do estado inicial ao objetivo, sendo composta por uma sequência de estados e operadores. Esta encapsula a sequência de passos, custo total e dimensão. Os métodos aqui existentes são:

- O construtor, que recebe como parâmetro o nó final da sequência e guarda-o num atributo privado. É criada uma lista vazia para armazenar os passos da sequência e criou-se uma variável *no* com o valor do nó final. Aqui também é reconstruído o caminho solução a partir do nó final, percorrendo os antecessores, e a cada passo, insere-se o mesmo no início da lista de passos, recriando corretamente os passos até à solução;
- Existem os métodos ***__iter__()*** e ***__getitem__(index)***, que, respetivamente, servem para iterar pela lista de passos e obter um determinado passo;
- Foram concebidas as propriedades para determinar a dimensão (número de transições, ou seja, a profundidade do nó final) e o custo (custo total do percurso)

Por fim, criou-se então a classe de dados *PassoSolucao*, que contém dois atributos, um estado e um operador, que representam, respetivamente, o estado atingido neste passo da solução e o operador que levou a este estado a partir do anterior. Esta classe de dados representa um passo individual na solução.

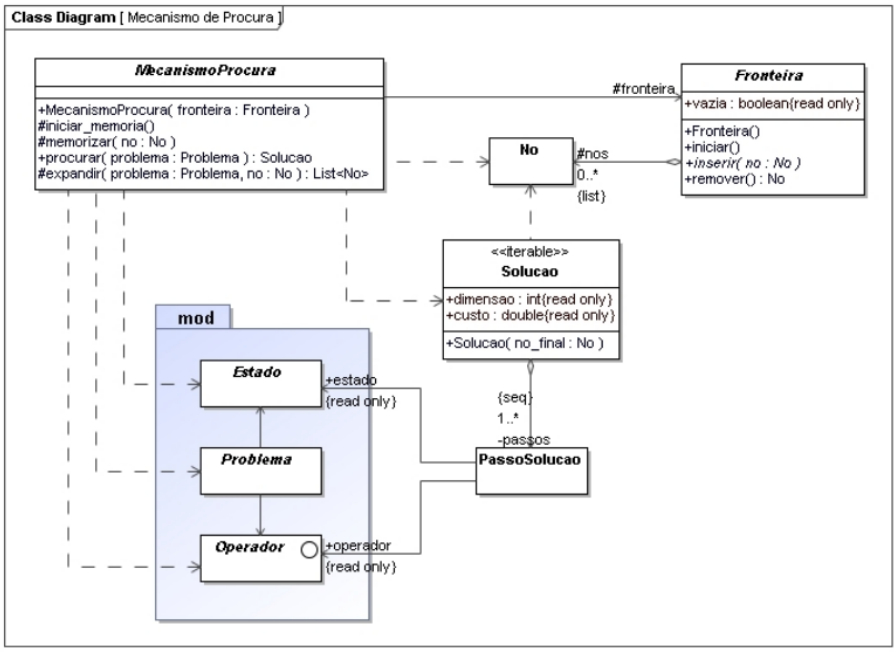


Figura 27 - Diagrama de classes do mecanismo de procura

Fronteira de Procura

A **Fronteira de Procura** é um conceito central nos algoritmos de procura. Ela representa a estrutura de dados responsável por armazenar todos os nós gerados que ainda não foram explorados (ou expandidos) durante a resolução de um problema. A forma de como a fronteira organiza e seleciona os próximos nós a expandir determina a estratégia de procura utilizada, como procura em largura (FIFO), procura em profundidade (LIFO) ou outras. Assim, a fronteira controla a ordem de exploração dos caminhos possíveis, influenciando diretamente a eficiência, completude e otimização do algoritmo de procura.

Neste tópico, vai ser abordada a implementação da ***FronteiraLIFO*** e ***FronteiraFIFO***, pois a classe *Fronteira* já foi abordada.

A *FronteiraLIFO* é uma fronteira onde o último nó inserido é o primeiro a ser removido (***L**ast **I**n **F**irst **O**ut*), enquanto *FronteiraFIFO* é uma fronteira onde o primeiro nó a ser inserido é o primeiro a ser removido (***F**irst **I**n **F**irst **O**ut*). Por isso, sendo que estas classes herdam de *Fronteira*, o único método a desenvolver é o método *inserir(no)*, pois é o único método abstrato presente na superclasse. Então, este método insere, na *FronteiraLIFO*, o nó na primeira posição da lista de nós, e na *FronteiraFIFO*, insere o nó na última posição lista dos nós.

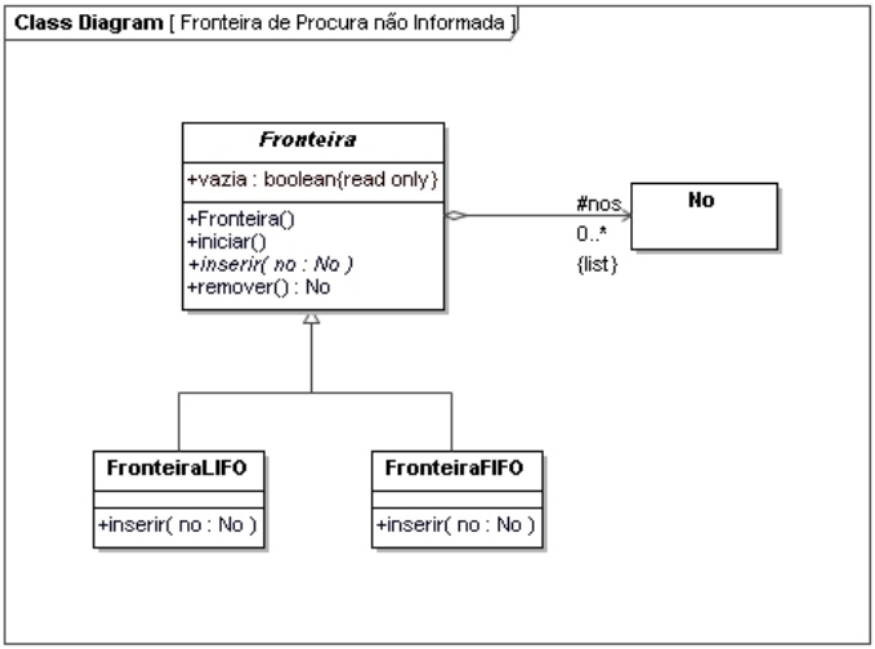


Figura 28 - Digrama de classes da fronteira de procura não informada

Mecanismo de Procura Não Informada

Os mecanismos de procura não informada são algoritmos de exploração do espaço de estados que não utilizam qualquer conhecimento específico sobre o objetivo além da definição do próprio problema.

Estes tipos de mecanismo utilizam fronteiras *LIFO* e *FIFO*, respetivamente, a procura em profundidade e procura em largura (que utiliza procura em grafo).

Para isso, foram implementadas as classes ***ProcuraProfundidade***, ***ProcuraGrafo*** e ***ProcuraLargura***.

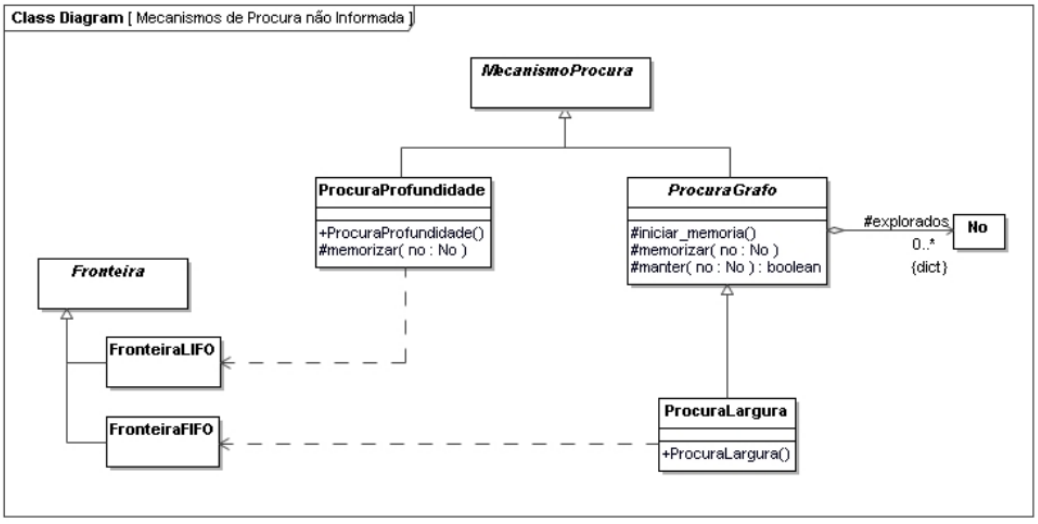


Figura 29 - Diagrama de classes dos Mecanismos de Procura não Informada

Procura em Profundidade

Esta parte do trabalho foi implementada de acordo com o que foi explicado em: V. Métodos de Procura Não Informada.

Primeiramente, definiu-se a classe *ProcuraProfundidade*, que representa, como o próprio nome indica, a procura em profundidade. Esta herda de *MecanismoProcura* e chama, no seu construtor, o construtor da superclasse passando como parâmetro (fronteira) a fronteira *LIFO*.

Depois, implementaram-se as classes ***ProcuraProfLim*** e ***ProcuraProfIter***, que respetivamente representam a procura em profundidade limitada e a procura em profundidade iterativa.

Em *ProcuraProfLim*, o construtor permite definir a profundidade máxima de exploração, limitando assim o número de níveis que o algoritmo pode explorar. O método principal, o ***_expandir***, garante que apenas os nós cuja profundidade seja inferior ao limite definido são efetivamente expandidos, devolvendo uma lista de sucessores nesses casos; caso contrário, retorna uma lista vazia, impedindo a expansão para além da profundidade máxima.

Já na *ProcuraProfIter*, o construtor recebe a profundidade máxima inicial, preparando a classe para realizar procuras limitadas. O método principal, ***procurar***, executa sucessivas procuras em profundidade limitada, aumentando progressivamente o limite de profundidade a cada iteração, até encontrar uma solução ou atingir o limite máximo especificado. Em cada ciclo, utiliza o método *procurar* da superclasse para tentar resolver o problema com o limite atual, retornando imediatamente a solução se for encontrada.

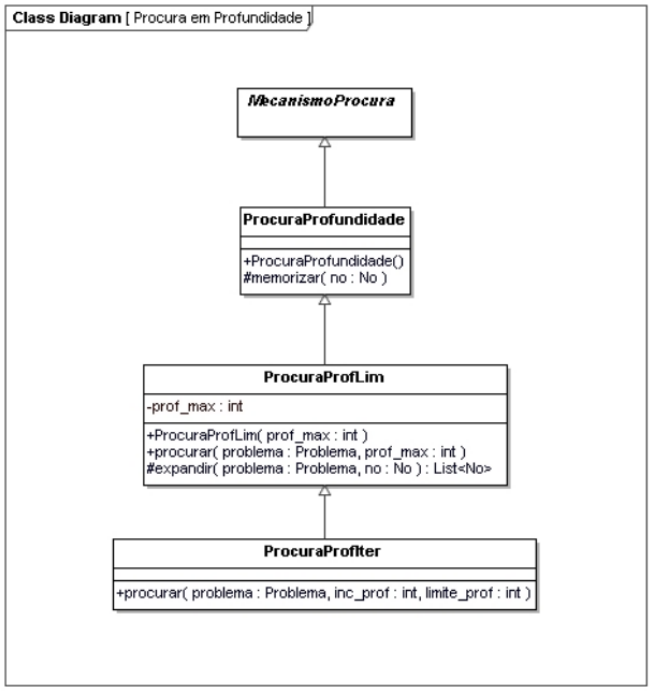


Figura 30 - Diagrama de classes da Procura em Profundidade

Procura em Grafo e Procura em Largura

No caso da **Procura em Grafo**, implementou-se a classe abstrata *ProcuraGrafo*, que especializa de *MecanismoProcura*, evitando a expansão de estados repetidos através de uma memória de nós explorados. O método ***_iniciar_memoria*** inicializa tanto a fronteira como um dicionário para guardar os estados já visitados, enquanto o método ***_memorizar*** só insere um nó na fronteira e na memória de explorados se este passar no critério definido pelo método abstrato ***_manter***, que deve ser implementado pelas subclasses para decidir se um nó deve ser mantido ou descartado.

Para a **Procura em Largura**, definiu-se a classe *ProcuraLargura*, que herda de *MecanismoProcura*. O construtor inicializa o mecanismo de procura com uma fronteira do tipo FIFO, utilizando a classe *FronteiraFIFO*, o que garante que os nós mais antigos são sempre expandidos primeiro. Esta abordagem assegura a exploração exaustiva de todos os nós de cada nível antes de avançar para níveis mais profundos, sendo completa e ótima, embora consuma muita memória.

Fronteira de Procura com Prioridade

A **fronteira com prioridade** é utilizada em algoritmos de procura informada, onde os nós são ordenados dinamicamente conforme a função de avaliação (que irá ser falada mais à frente).

A classe ***FronteiraPrioridade*** especializa da classe *Fronteira*, implementando o método ***inserir***, que calcula a prioridade de cada nó usando o avaliador e insere-o numa estrutura de dados eficiente para listas prioritárias, e o método ***remover***, onde é retirado e devolvido o nó com maior prioridade. Assim, a classe garante que a expansão dos nós segue sempre a ordem definida pela função de avaliação, otimizando a procura guiada por heurísticas.

Foi também implementada a interface ***Avaliador***, que é utilizada em algoritmos de procura informada para calcular a prioridade de expansão de cada nó na fronteira. Um avaliador implementa uma função de avaliação (como f(n) = g(n), f(n) = h(n) ou f(n) = g(n) + h(n)), que pode considerar o custo acumulado, uma heurística ou ambos, determinando assim a ordem pela qual os nós são processados.

A interface *Avaliador* contém o método abstrato ***prioridade(no)***, que é responsável por calcular a prioridade de cada nó a ser expandido. Este método é concretamente implementado de acordo com o avaliador específico.

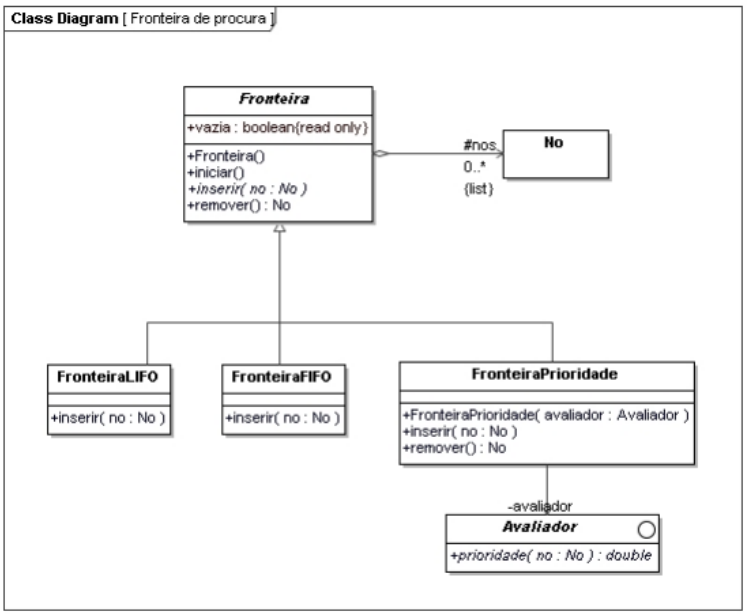


Figura 31 - Diagrama de classes da fronteira de procura com prioridade

Procura Melhor-Primeiro

A classe abstrata ***ProcuraMelhorPrim*** herda de *ProcuraGrafo* e foi implementada de acordo com a teoria explicada anteriormente. Os métodos aqui presentes são:

- Construtor: recebe um objeto avaliador responsável por calcular a prioridade dos nós e inicializa a superclasse com uma *FronteiraPrioridade* configurada com esse avaliador, garantindo que os nós mais promissores são explorados primeiro;
- ***_manter***: determina se um nó deve ser mantido na fronteira, permitindo apenas a inserção de nós que representem estados ainda não explorados ou que apresentem um custo menor do que o anteriormente registado para esse estado

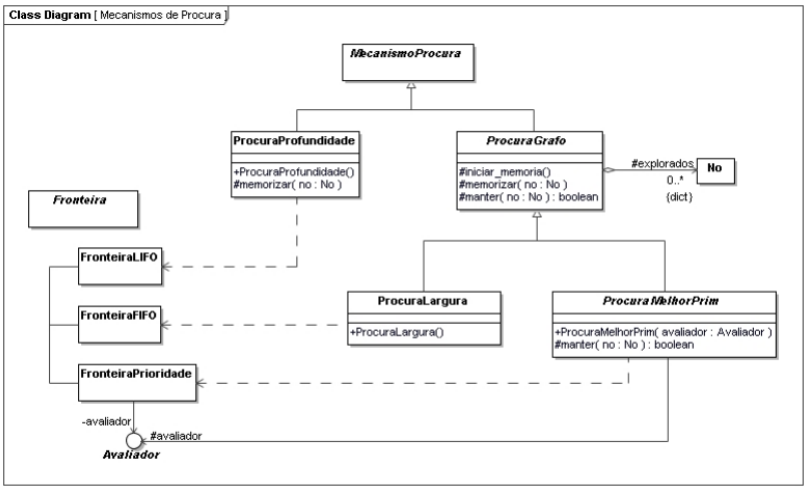


Figura 32 - Diagrama de classes de mecanismos de procura incluindo a Procura Melhor-Primeiro

Procura de Custo Uniforme

De acordo com o que o foi explicado na teoria, a classe ***ProcuraCustoUnif*** especializa de ***ProcuraMelhorPrim*** e foi implementado o seguinte:

- Construtor: cria uma instância de ***AvaliadorCustoUnif***, responsável por calcular a prioridade dos nós com base apenas no custo acumulado (f(n) = g(n)), e passa este avaliador para a superclasse, que por sua vez utiliza uma fronteira de prioridade para garantir a ordem correta de expansão

Para o correto funcionamento da classe anterior, foi necessário criar, como já foi mencionado, a classe *AvaliadorCustoUnif*, que utiliza apenas o custo acumulado do nó (g(n)) para determinar a prioridade de expansão. O único método aqui realizado é *prioridade(no)*, que devolve o custo acumulado do nó passado como parâmetro.

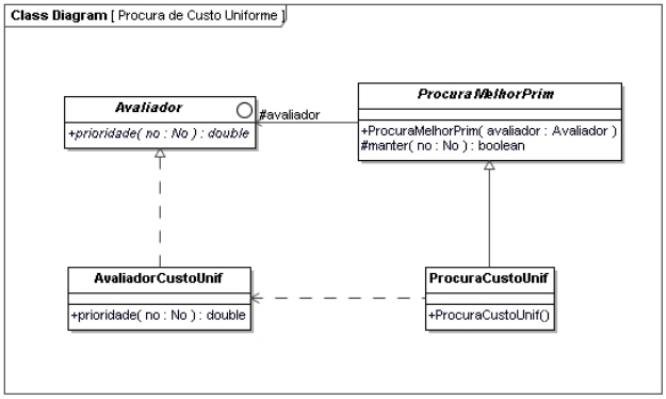


Figura 33 - Diagrama de classes da Procura de Custo Uniforme

Procura Informada e Avaliadores de Nós

Para completar as procuras que utilizam a melhor-primeiro, criaram-se, de acordo com a teoria, as seguintes classes:

- ***ProcuraInformada***: é uma classe abstrata que estende de *ProcuraMelhorPrim*, cujo principal método é o ***procurar(problema, heurística),*** que recebe um problema e uma heurística como parâmetros. Este método configura a heurística no avaliador interno antes de delegar a execução da procura ao método da superclasse;
- ***Heuristica***: classe abstrata que define a interface para funções heurísticas. O método abstrato ***h(estado)*** é implementado por subclasses e serve para calcular uma estimativa do custo mínimo restante desde um estado até ao objetivo, retornando um valor não-negativo;
- ***ProcuraAA***: esta classe implementa o algoritmo procura A*, herdando de *ProcuraInformada*. O construtor inicializa o mecanismo de procura com o avaliador A* (***AvaliadorAA***), que calcula a função de avaliação f(n) = g(n) + h(n), onde g(n) é o custo acumulado até ao nó n e h(n) é a estimativa heurística do custo até ao objetivo;
- ***ProcuraSofrega***: especializando também de *ProcuraInformada*, esta classe implementa o algoritmo de Procura Sôfrega. Primeiramente, o construtor inicializa o mecanismo de procura com o avaliador sôfrega (***AvaliadorSofrega***), que utiliza apenas a função heurística h(n) para priorizar a exploração dos nós, ignorando o custo acumulado g(n);

Estas procuras e função heurística precisam de avaliadores específicos para o seu correto funcionamento, como já foi dito na lista anterior.

Os avaliadores são então componentes fundamentais, pois determinam a prioridade com que os nós são explorados durante a busca. A classe abstrata ***AvaliadorHeur*** serve de base para avaliadores que utilizam funções heurísticas, permitindo a configuração dinâmica da heurística a ser usada. A partir dela, derivam-se avaliadores concretos como o ***AvaliadorAA***, que implementa a função de avaliação do algoritmo A* (f(n) = g(n) + h(n)), combinando o custo acumulado e a estimativa heurística, e o ***AvaliadorSofrega***, que prioriza apenas o valor heurístico h(n), caracterizando a procura sôfrega.

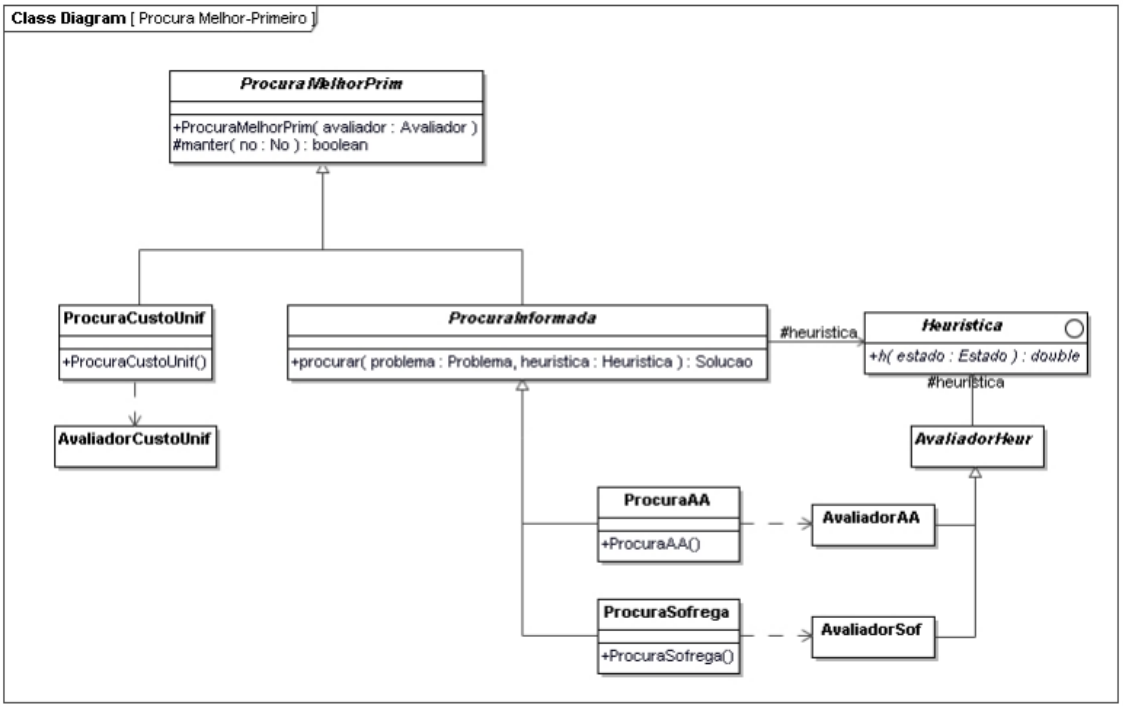


Figura 34 - Diagrama de classes da Procura Melhor-Primeiro atualizada

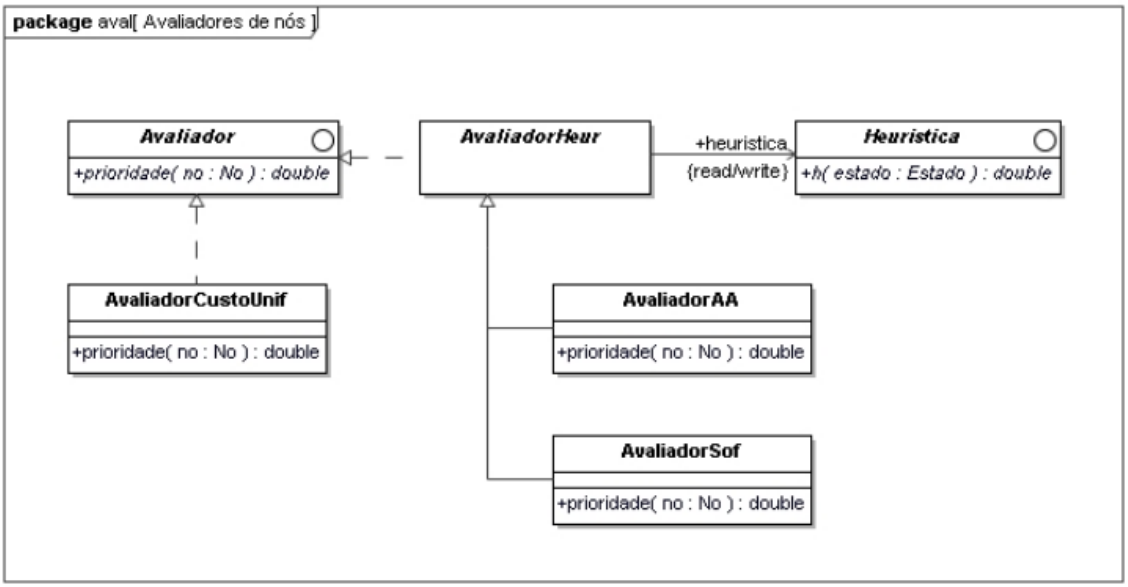


Figura 35 - Diagrama de classes de todos os avaliadores de nós e heurística

Problema de Contagem

Como neste ponto ainda não se tinha o agente deliberativo desenvolvido, foi criada uma *package* ***contagem***, que por sua vez contém a package ***modelo***, onde foram definidas as classes ***EstadoContagem***, ***HeuristicaContagem***, ***OperadorIncremento*** e ***ProblemaContagem***, de modo a testar o que foi implementado anteriormente.

A classe *EstadoContagem* é uma implementação concreta da classe abstrata Estado, representando um estado numérico contável. O seu construtor recebe e armazena um valor numérico que identifica unicamente cada estado, funcionando como o seu identificador. O método ***id_valor()*** devolve esse valor, implementando o método abstrato da superclasse e permitindo a comparação e o *hashing* de estados.

Já a classe *HeuristicaContagem* implementa uma heurística específica para problemas de contagem, onde o objetivo é atingir um valor final a partir de um valor inicial por meio de incrementos. O construtor recebe e armazena o valor objetivo final, necessário para o cálculo da heurística. O método principal, *h(estado)*, calcula a estimativa heurística do custo restante como a distância absoluta entre o valor atual do estado e o valor final, ou seja, $|valor\_final - valor\_atual|$.

A *OperadorIncremento* é uma implementação da interface *Operador*, destinada a manipular estados numéricos em problemas de contagem. O construtor recebe e armazena um valor de incremento, que será utilizado em todas as operações. O método *aplicar(estado)* soma esse incremento ao valor do estado atual, criando e devolvendo um novo objeto *EstadoContagem* com o valor atualizado, seguindo o princípio de que operadores geram estados sucessores a partir de um estado dado. Já o método *custo(estado, estado_suc)* calcula o custo da transição como o quadrado do incremento, penalizando incrementos maiores e implementando uma métrica de custo quadrática, conforme sugerido nos materiais teóricos sobre operadores e custos de transição.

Por fim, a classe *ProblemaContagem* modela um problema de contagem numérica, especializando a classe abstrata *Problema*. O método *__init__* recebe um valor inicial, um valor final e uma lista de incrementos, criando o estado inicial como um objeto *EstadoContagem* e gerando uma lista de operadores *OperadorIncremento* para cada incremento fornecido, além de armazenar o valor final como objetivo privado. O método *objectivo(estado)* verifica se o valor do estado atual é maior ou igual ao valor final, determinando se o objetivo foi atingido.

Para de facto se puder resolver um problema de contagem, foi criada a classe ***Contagem***, que serve como ponto de entrada para a definição, configuração e resolução de problemas de contagem numérica através de diferentes algoritmos de procura. Nesta classe, são definidos os parâmetros do problema, como o valor inicial, valor final e os incrementos possíveis (lista dos os algarismos do número de aluno), sendo depois instanciado um objeto *ProblemaContagem* que encapsula o estado inicial, os operadores e o objetivo. A classe permite testar várias estratégias de procura, como procura em profundidade, largura, custo uniforme, procura sôfrega e A*, bastando instanciar o mecanismo de procura desejado e, quando necessário, uma heurística adequada

(*HeuristicaContagem*). Os valores utilizados foram $valor_{inicial} = 0$; $valor_{final} = 20$; e $incrementos = [5, 1, 6, 9, 6]$.

Os resultados para cada teste foram:

Resultados	Profundidade	Profundidade Limitada	Largura	Profundidade Iterativa	Custo Uniforme	A*	Sôfrega
Dimensão	4	4	3	3	20	20	4
Custo	144	144	142	153	20	20	164
Caminho	0 → 6 → 12 → 18 (incremento de 6 em cada passo)	0 → 6 → 12 → 18 (incremento de 6 em cada passo)	0 → 5 → 11 → 20 (incrementos: 5, 6, 9)	0 → 6 → 12 → 21 (incrementos: 6, 6, 9)	0 → 1 → 2 → … → 20 (incremento de 1 em cada passo)	0 → 1 → 2 → … → 20 (incremento de 1 em cada passo)	0 → 9 → 18 → 19 → 20 (incrementos: 9, 9, 1, 1)
Nós processados	21	21	221	16	101	101	21
Nós em memória	21	21	221	15	81	81	17

Tabela 2 - Resultados para cada tipo de procura

A procura em profundidade (LIFO) segue sistematicamente o último ramo expandido, o que pode resultar na exploração de caminhos longos e sub-ótimos. No exemplo em análise, a estratégia selecionou repetidamente o incremento 6 (último da lista), não alcançando o objetivo (20), tendo terminado no valor 18. O custo associado é elevado, dado que consiste na soma de quatro incrementos de 6, cada um contribuindo com 6^2. Nesta abordagem, o número de nós processados é igual ao número de nós mantidos em memória, uma vez que não ocorre eliminação de nós durante o processo. Esta estratégia de procura não é adequada neste contexto, sendo nem completa nem garantidamente eficaz, pois pode falhar em encontrar uma solução na presença de ciclos ou caminhos infinitos.

No caso da procura em profundidade limitada, ao utilizar o limite de profundidade padrão (10), o resultado obtido coincide com o da procura em profundidade simples, uma vez que a profundidade da solução encontrada (4) é inferior ao limite imposto. Assim, o comportamento do algoritmo é, neste cenário, idêntico ao da versão não limitada.

A procura em largura (FIFO) privilegia a descoberta do caminho com menor profundidade, ou seja, com o menor número de passos, mas não garante a solução de menor custo, exceto quando todos os custos são uniformes. No exemplo em questão, foi identificado um caminho com três passos; contudo, este não corresponde ao caminho de menor custo possível (como demonstrado na procura de custo uniforme). Devido à

natureza da expansão em largura, o número de nós processados é significativamente elevado.

A estratégia de procura em profundidade iterativa combina as abordagens de profundidade limitada e de procura em largura, incrementando progressivamente o limite de profundidade a cada iteração. No caso em análise, identificou um caminho com três passos, embora com um custo superior ao obtido pela procura em largura, dado que esta estratégia não tem em consideração o custo das transições, apenas a profundidade. O número de nós processados é inferior, resultado da limitação imposta em cada iteração.

A procura de custo uniforme expande, em cada passo, o nó com menor custo acumulado desde a origem. Como o incremento de valor 1 apresenta o custo mínimo possível ($1^2 = 1$), o algoritmo tende a privilegiar sucessivos incrementos de 1, conduzindo a um caminho ótimo em termos de custo, embora composto por muitos passos (20 no total). O número de nós processados é elevado, devido à necessidade de explorar múltiplos caminhos alternativos, mas a solução obtida é ótima em custo.

O algoritmo A* conjuga o custo acumulado com uma estimativa heurística do custo restante até ao objetivo. Neste caso, foi utilizada uma heurística admissível e consistente — a distância absoluta ao objetivo. Como tal, o algoritmo A* encontra o mesmo caminho ótimo em custo que a procura de custo uniforme. O número de nós processados e a memória ocupada são equivalentes aos da procura de custo uniforme, dado que, nesta situação concreta, a heurística não altera a ordem de expansão dos nós.

Por fim, o algoritmo de procura sôfrega baseia-se exclusivamente na heurística para selecionar o próximo nó a expandir, desconsiderando o custo acumulado. Esta estratégia pode favorecer incrementos de maior valor (por exemplo, 9) para reduzir rapidamente a distância ao objetivo, mas tal pode originar soluções com custos totais elevados — como se verifica neste caso, em que o custo total ascende a $9^2 + 9^2 + 1 + 1 = 164$. Esta abordagem não garante soluções ótimas, embora possa conduzir rapidamente a uma qualquer solução.

Parte 4

Nesta parte, o foco recai na implementação de um **controlo deliberativo**, que inclui componentes como o **estado do agente**, os **operadores de movimento**, o **processamento deliberativo**, e o **planeamento automático**. Este planeamento pode ser realizado com base em diferentes abordagens, como **PEE (Procura em Espaço de Estados)** ou **PDM (Processo de Decisão de Markov)**, integrando-se com um **modelo do mundo** que fornece a base de conhecimento necessária para a tomada de decisões.

Arquitetura de Agentes Deliberativos

A arquitetura deliberativa distingue-se pela sua capacidade de prever cenários futuros e planear ações de forma otimizada, ultrapassando as limitações das abordagens reativas. Contudo, requer um elevado poder computacional e mecanismos eficazes para operar em ambientes dinâmicos. É especialmente adequada para contextos onde a tomada de decisões estratégicas é fundamental, como na robótica autónoma ou em sistemas de gestão inteligente. Neste tipo de arquitetura, a memória assume um papel central, sustentando os processos de deliberação e tomada de decisão, ao armazenar e processar informações relevantes do domínio do problema.

I. Tempo e Comportamento

O comportamento dos agentes está fortemente ligado à forma como estes lidam com as diferentes dimensões temporais: presente, passado e futuro. Agentes puramente reativos operam no presente, respondendo apenas a estímulos imediatos sem considerar o histórico ou as consequências futuras. Agentes reativos com memória têm em conta o passado, utilizando experiências anteriores para guiar as suas respostas. Já os agentes deliberativos distinguem-se por anteciparem o futuro, simulando possíveis estados e selecionando ações com base em previsões, adotando assim um comportamento orientado à antecipação.

II. Memória e Comportamento

A memória é um elemento essencial para suportar comportamentos que envolvem diferentes horizontes temporais. Quando ligada ao passado, a memória permite aos agentes repetir ações bem-sucedidas ou evitar erros anteriores. No presente, suporta a perceção e a reação imediata ao ambiente. Em relação ao futuro, a memória permite simular cenários possíveis e planear ações, sendo esta capacidade fundamental para os agentes deliberativos.

III. Raciocínio Automático

O raciocínio automático é uma componente central nas arquiteturas deliberativas, permitindo ao agente explorar e avaliar diferentes opções de ação. A exploração de opções envolve a antecipação de eventos e a simulação interna do ambiente, exigindo representações detalhadas do domínio do problema. Já a avaliação de opções é feita com base em critérios como custo, recursos disponíveis e utilidade esperada, permitindo ao agente selecionar a ação mais vantajosa de forma racional e fundamentada.

IV. Raciocínio Prático

O raciocínio prático é um processo orientado para a ação, que utiliza representações simbólicas dos objetivos, do ambiente e das ações possíveis. Este processo divide-se em duas fases principais: a deliberação, onde o agente decide o que pretende alcançar, resultando na definição de objetivos, e o planeamento, onde é decidido como alcançar esses objetivos, gerando planos de ação concretos. Os produtos finais deste raciocínio são planos que orientam diretamente o comportamento do agente.

V. Processo de Tomada de Decisão e Ação

A tomada de decisão nas arquiteturas deliberativas decorre de um ciclo estruturado. Primeiro, o agente observa o mundo e gera perceções com base no ambiente. Em seguida, atualiza o seu modelo interno, incorporando essas novas informações. Com base nesse modelo, delibera sobre os objetivos a seguir e planeia as ações necessárias para os atingir. Finalmente, executa o plano elaborado. Contudo, este processo enfrenta desafios significativos, como a limitação de recursos computacionais e a imprevisibilidade do ambiente. Para lidar com estas dificuldades, o ciclo pode incluir uma fase de reconsideração, onde o agente reavalia os objetivos e planos em função de mudanças ocorridas, garantindo uma adaptação contínua e eficaz.

VI. Controlo Deliberativo

O controlo deliberativo organiza os componentes do raciocínio prático e do modelo do mundo de forma modular, facilitando a gestão eficiente do comportamento do agente. Este controlo coordena a receção e interpretação das perceções, a atualização do modelo interno e a geração de ações através do ciclo de decisão. Com esta estrutura, o agente consegue adaptar-se de forma consistente aos seus objetivos, mesmo em ambientes dinâmicos e complexos, assegurando uma resposta estratégica e informada às circunstâncias em constante mudança.

Planeamento Automático

O planeamento automático é descrito como um processo deliberativo cujo objetivo é gerar planos de ação (sequências de ações) para alcançar objetivos previamente definidos. Este processo é uma componente central do raciocínio prático abordado anteriormente, especificamente na fase de planeamento (decidir como fazer após a deliberação definir o que fazer).

Este planeamento é feito com base em técnicas de raciocínio automático, como a procura em espaço de estados (falada anteriormente), ou processos de decisão de Markov (irá ser falado mais à frente).

O processo de planeamento exige a existência de um modelo que inclua a definição do estado inicial do problema, o conjunto de estados válidos (essencial, por exemplo, em processos de decisão de Markov), e um conjunto de operadores que descrevem as possíveis ações. Além disso, é necessário especificar claramente os objetivos a atingir, de forma a orientar o planeamento das ações mais adequadas.

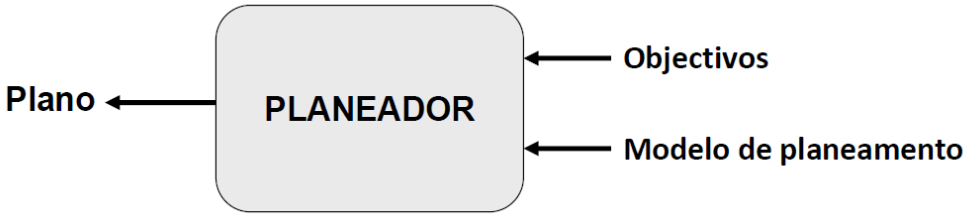


Figura 36 - Funcionamento do Planeador

O raciocínio automático por procura resolve problemas de planeamento com base numa abstração do domínio designada espaço de estados, onde cada configuração possível é representada como um estado, e as ações que transformam uma configuração noutra são representadas por operadores. Aplicar um operador origina uma transição de estado, cada uma com o seu custo, gerando novos estados. Assim, a solução do problema consiste num percurso no espaço de estados, ou seja, numa sequência de estados e operadores que liga o estado inicial ao estado objetivo (plano de ação).

Mecanismo de Planeamento de um Agente

Numa arquitetura de agente autónomo, o planeador elabora planos de ação a partir dos objetivos definidos pelo processo de deliberação. Quando esse planeamento é feito através de procura em espaços de estados, é essencial ter em conta o modelo do problema, a heurística (caso seja aplicável) e o mecanismo de procura que orienta a navegação no espaço de estados.

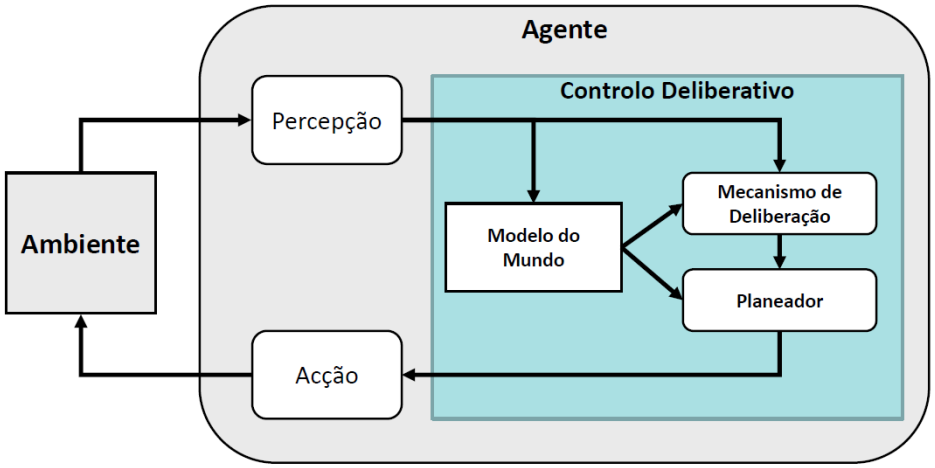


Figura 37 - Esquema do mecanismo de planeamento de um agente

Processos de Decisão Sequencial

Os **processos de decisão sequencial (PDS)** referem-se a situações em que **o valor de uma ação não depende apenas do estado atual**, mas de **uma sequência de ações ao longo do tempo**, cujos resultados podem ser **não determinísticos**. Este tipo de problema é comum em ambientes dinâmicos, como a navegação de veículos autónomos, onde a incerteza surge da impossibilidade de obter informação completa sobre o domínio do problema.

Ao contrário de problemas de decisão estáticos, onde uma única ação conduz a um resultado imediato e determinístico, os PDS envolvem uma **sequência de decisões**, cada uma influenciando a evolução do sistema e as possibilidades futuras. Esta complexidade deve-se a três características fundamentais:

- **Recompensas diferidas**:
  - As consequências das ações tomadas por um agente não se concretizam imediatamente. Em vez disso, os efeitos vão-se acumulando ao longo do tempo, sendo necessário avaliar o impacto total de uma sequência de ações;
- **Incerteza**:
  - A execução de uma ação num determinado estado não garante uma transição única para um estado seguinte. Existem **probabilidades associadas às transições**, refletindo a incerteza do ambiente ou a imprevisibilidade de certos fatores. Esta propriedade implica que o agente deve planear tendo em conta múltiplos cenários possíveis;
- **Espaço de estado não-determinista**:
  - O sistema é modelado como um conjunto de estados possíveis, e cada ação tomada num estado pode originar uma transição probabilística para um ou mais estados seguintes. Estas transições são formalmente representadas por uma função de transição de estados $T(s, a, s')$, enquanto os ganhos ou perdas são modelados por uma função de recompensa $R(s, a, s')$

Exemplo:

Estados = $\{s_0, s_1, s_2\}$

Acções = $\{a_0, a_1\}$

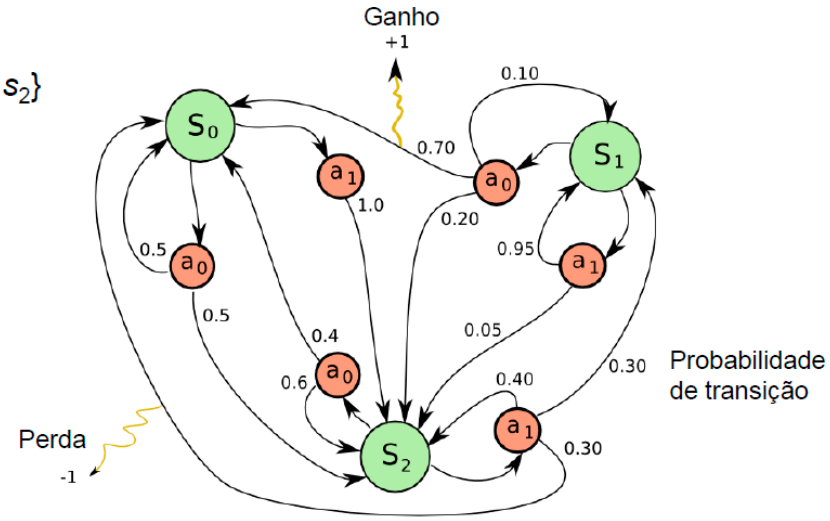


Figura 39 - Exemplo de um espaço de estados não determinista

Propriedade de Markov

A **propriedade de Markov**, introduzida por Andrey Markov, diz que um processo estocástico tem esta propriedade se os **estados futuros** dependem apenas do **estado presente**, sendo independentes dos estados passados. Isso simplifica a modelação, pois:

- A previsão dos estados sucessores baseia-se apenas no estado atual;
- O processo é representado como uma **cadeia de Markov**, onde a evolução é descrita por uma sequência de **estado** -> **ação** -> **recompensa**

Processos de Decisão de Markov

Os PDM estruturam a tomada de decisão em contextos de incerteza através das seguintes componentes fundamentais:

- **Conjunto de estados S**: engloba todas as possíveis configurações em que o sistema pode se encontrar;
- **Conjunto de ações $A(s)$**: conjunto de ações disponíveis quando o sistema de encontra num determinado estado $s \in S$;
- **Modelo de transição $T(s, a, s')$**: probabilidade de o sistema transitar do estado s para o estado s' ao aplicar uma ação a;
- **Modelo de recompensa $R(s, a, s')$**: representa o valor esperado da recompensa associada à transição de s para s' ao aplicar a ação a. Em diferentes contextos, pode ser expressa como:
  - $R(s, a, s')$: caso geral;
  - $R(s, a)$: recompensa depende apenas do estado e da ação;
  - $R(s)$: depende só do estado;

- **Tempo discreto** $t = 0, 1, 2, \ldots$: as decisões são tomadas em instantes discretos, com o sistema a evoluir passo a passo ao longo do tempo

Exemplo:

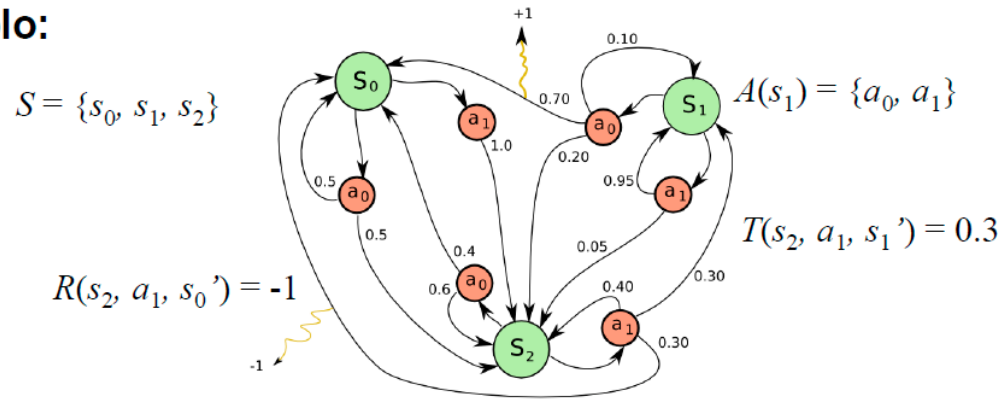


Figura 40 - Exemplo da representação do mundo sob a forma de PDM

Tomada de Decisão Sequencial e Utilidade

A tomada de decisão sequencial requer uma **medida de desempenho** chamada **utilidade** $\boldsymbol{U(s)}$, que reflete o valor a longo prazo de um estado, considerando o efeito cumulativo das recompensas ao longo de uma sequência de estados e ações.

Qual é a diferença entre a recompensa e a utilidade?

- A recompensa é um valor de curto prazo atribuído ao resultado imediato de uma ação. Este valor pode ser positivo (ganho) ou negativo (perda);
- Já a utilidade é um valor de longo prazo que reflete o valor acumulado de um estado ao longo de uma sequência de estados futuros

A utilidade calcula-se somando as recompensas de cada estado. Pode é acontecer as recompensas não estarem limitadas a uma gama finita de valores, tornando a soma infinita e impossibilita a decisão. Além disso, não considera o efeito da passagem do tempo discreto.

Para isso, atribuiu-se a cada recompensa um **fator de desconto** $\boldsymbol{\gamma} \in [\mathbf{0}, \mathbf{1}]$, que reduz o peso das recompensas futuras, refletindo a preferência por ganhos imediatos e evitando somas infinitas. Utilizando o exemplo da página 18 do pdf *15-pds.*pdf, para $\gamma = 0.5$, a ação a_2 tem maior utilidade do que a_1, indicando que a_2 é a decisão racional, isto é, maximiza a utilidade.

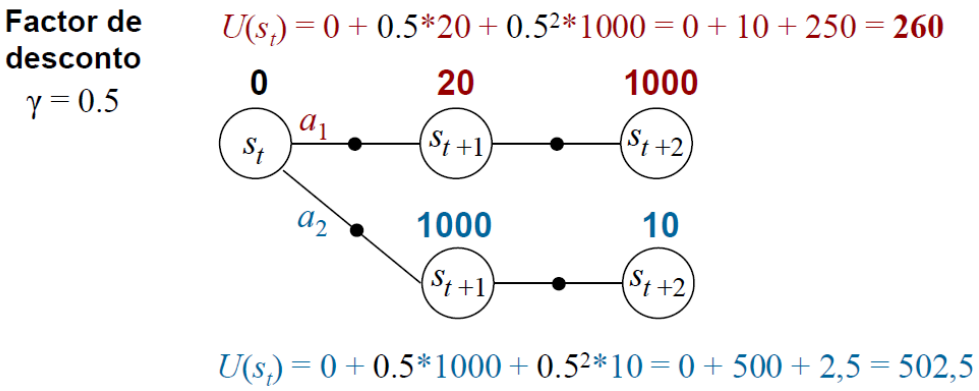


Figura 41 - Exemplo do cálculo da utilidade utilizando o fator de desconto

Política de Tomada de Decisão

A **política** ($\boldsymbol{\pi}$) é uma função que define a estratégia de ação de um agente, especificando qual ação realizar (ou a probabilidade de realizá-la) em cada estado, visando maximizar a utilidade esperada. Esta pode ser:

- **Política determinista**: para cada estado s, escolhe uma ação específica $\pi: S \rightarrow A(s)\,; s \in S$;
- **Política não determinista**: associa probabilidades a ações em cada estado $\pi: S \times A(s) \rightarrow [0,1]\,; s \in S$

Para se poder encontrar uma solução ótima, é necessário calcular a utilidade através da Equação de Bellman, que utiliza modelos de transição e recompensa probabilísticos, e definir uma política ótima que selecione, para casa estado, a ação que maximiza o valor esperado a longo prazo, considerando o fator de desconto para garantir convergência.

$$U^{\pi}(s) = \mathrm{E}\left\langle r_1 + \gamma r_2 + \gamma^2 r_3 + \ldots \right\rangle \qquad \textbf{Equação de Bellman}$$
$$= \mathrm{E}\left\langle r_1 + \gamma U^{\pi}(s') \right\rangle$$
$$= \sum_a \pi(s,a) \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma U^{\pi}(s') \right]$$

Figura 42 - Equação de Bellman

O cálculo da política ótima é feito da seguinte maneira: para cada estado s, escolhe-se a ação a com maior utilidade, ou seja:

$$\pi * (s) = arg\ max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma U_i(s')], \forall_s \in S$$

Finalmente, para calcular iterativamente a utilidade, é utilizada a Equação de Bellman. O algoritmo é o seguinte:

Iniciar $U(s)$:

$$U(s) \leftarrow 0, \forall_S \in S$$

Iterar $U(s)$:

$$U_{i+1}(s) \leftarrow \ max_a \ \sum_{s'} \ T(s,a,s')[\,R(s,a,s') + \gamma U_i(s')], \forall_s \in S$$

No limite:

$$U \rightarrow U^{\pi *}$$

Implementação

Controlo Deliberativo

Para começar a implementar esta última parte do projeto, começou-se por definir a *package* ***controlo_delib***, segundo a teoria anteriormente explicada. Esta *package* contém as classes ***ControloDelib***, ***MecDelib***, ***EstadoAgente***, ***ModeloMundo*** e ***OperadorMover***.

A classe *ControloDelib* é o "cérebro" deliberativo do agente, responsável por decidir **o que fazer** (deliberação), **como fazer** (planeamento) e **executar** ações, com base nas perceções do ambiente. Implementa o ciclo deliberativo clássico: observar, atualizar, deliberar, planear e executar. Os métodos presentes nesta classe são:

- ***__init__(planeador)***: construtor da classe que inicializa o modelo de mundo, utilizando uma instância de *ModeloMundo*, o mecanismo de deliberação, passando como parâmetro o modelo de mundo anteriormente inicializado, armazena o planeador num atributo privado, e inicia uma lista de objetivos e um plano que inicialmente estão vazios;
- ***processar(percepcao)***: este método é o responsável por realizar ações perante a perceção obtida, considerando o estado do modelo do mundo. É aqui que o ciclo deliberativo é implementado, onde o agente assimila a perceção; se necessário, reconsidera e só depois delibera e planeia; e finalmente executa a ação. Este ciclo depende dos métodos internos ***__assimilar***, ***__reconsiderar***, ***__deliberar***, ***__planear*** e ***__executar***;
- ***__assimilar(percepcao)***: atualiza o modelo do mundo, com base nas perceções do agente;
- ***__reconsiderar()***: determina se é necessário refazer o planeamento com base no estado atual do ambiente, percebendo se o mesmo foi alterado ou não;
- ***__deliberar()***: usando o mecanismo de deliberação, este método gera novos objetivos com base no ambiente atual;
- ***__planear()***: gera um novo plano de ação para atingir os objetivos atuais, utilizando o planeador. Caso haja objetivos, é criado um plano tendo em conta o modelo de mundo e o os objetivos, caso não haja, não é criado um plano;

- ***__executar()***: executa o próximo passo do plano, retornando a ação a realizar, ou seja, o agente age no ambiente de acordo com o plano. O método verifica se o plano é válido, obtém o estado atual, recupera o operador e, caso haja uma ação válida, retorna a ação associada ao operador, se não, invalida o plano;
- ***__mostrar()***: serve apenas para exibir o estado interno do controlo numa interface gráfica

A classe ***MecDelib*** representa o **mecanismo de deliberação** de um agente deliberativo. É responsável por analisar o modelo do mundo e decidir **quais são os objetivos do agente**, ou seja, "o que fazer a seguir". No ciclo deliberativo, **deliberar** significa escolher os fins (objetivos) a alcançar, antes de planear como os atingir. As funções existentes são:

- O construtor, onde é guardado o modelo de mundo passado como parâmetro num atributo privado. Segundo a arquitetura deliberativa, o mecanismo de deliberação precisa de aceder ao conhecimento do ambiente (modelo do mundo) para poder raciocinar sobre os objetivos;
- ***__deliberar()***: este é o método principal, que coordena o processo de geração e seleção de objetivos;
- ***__gerar_objetivos()***: responsável por gerar uma lista de todos os estados do modelo do mundo que sejam alvos. Este itera por todos os estados do modelo do mundo e filtra apenas os estados cujo elemento é um alvo;
- ***__selecionar_objetivos(objetivos)***: ordena a lista de objetivos por proximidade ao agente, ou seja, os objetivos que estiverem mais perto. Durante o processo de raciocínio automático, existem dois tipos de atividades principais, a exploração das melhores opções possíveis e a avaliação das mesmas, percebendo quais são as melhores. Na avaliação de opções, é avaliado o custo, sendo o custo melhor quanto menor for. Isso significa que a distância até a um alvo também tem que ser menor

A classe seguinte que foi implementada foi *EstadoAgente*, herdando de *Estado*. Esta representa o **estado interno do agente** no ambiente, ou seja, a sua posição (um par de coordenadas (x, y)):

- No construtor, é inicializada e guardada a posição específica do agente num atributo privado;
- ***id_valor()***: gera um identificador único para o estado, usando o valor hash da posição;
- Propriedade ***posição()***: permite aceder à posição do agente de forma "read-only", ou seja, serve só para leitura e garante que não há maneiras de alterar este atributo fora da classe;

Posteriormente, criou-se a classe *ModeloMundo*, que, como o próprio nome indica, contém a representação interna do ambiente. Esta classe especializa de *ModeloPlan*, que irá ser explicada mais à frente. A classe foi desenvolvida com os seguintes métodos:

- ***__init__()***: inicializa todos os atributos do modelo do mundo, sendo eles o estado do agente, os estados válidos no ambiente, um dicionário de elementos (alvo, obstáculo, ...) presentes em cada posição, uma lista de operadores de movimento disponíveis e se o modelo foi alterado desde a última atualização;
- ***obter_estado(), obter_estados(), obter_operadores(), obter_elemento()***: respetivamente, retornam o estado atual do agente no ambiente, a lista de todos os estados válidos do ambiente, a lista de operadores disponíveis e o elemento presente numa determinada posição;
- ***distancia(estado)***: calcula a distância euclidiana entre o estado atual do agente e outro estado. É utilizado para avaliar custos de movimento, priorizar objetivos e planear caminhos;
- ***actualizar(percepcao)***: atualiza o modelo do mundo com base em novas informações percebidas pelo agente. O método atualiza a posição do agente, compara os elementos percebidos com os atuais para detetar mudanças, e se houver mudanças, atualiza os elementos e reconstrói a lista de estados e marca o modelo como alterado
- ***mostrar(vista)***: mostra o estado atual do modelo numa interface gráfica;
- Propriedades ***elementos()*** e ***alterado()***, que respetivamente retornam os elementos do modelo do mundo e se o modelo foi alterado;
- Foi criado também o método ***__contains__(estado)*** que permite usar o operador ***in*** para verificar se um estado pertence à lista de estados válidos

Por fim, definiu-se a classe *OperadorMover*, que, especializando da interface *Operador*, representa um operador de movimento para agentes em ambientes 2D discretos. Cada instância corresponde a uma ação de deslocamento numa direção específica, definida por um ângulo e uma ação concreta (*Accao*). O método ***aplicar*** tenta gerar um novo estado do agente ao mover-se na direção indicada, validando se o novo estado é permitido no modelo do mundo; se for válido, retorna o novo estado. O método ***custo*** calcula o custo da transição entre estados, normalmente a distância euclidiana entre posições. O método privado ***__translacao*** realiza o cálculo geométrico da nova posição com base no passo e ângulo da direção. Teoricamente, esta classe implementa o conceito de operador como uma ação que gera transições de estado, fundamental para o planeamento deliberativo, permitindo ao agente simular e avaliar movimentos possíveis no ambiente.

Toda esta implementação foi concretizada de acordo com os seguintes diagramas:

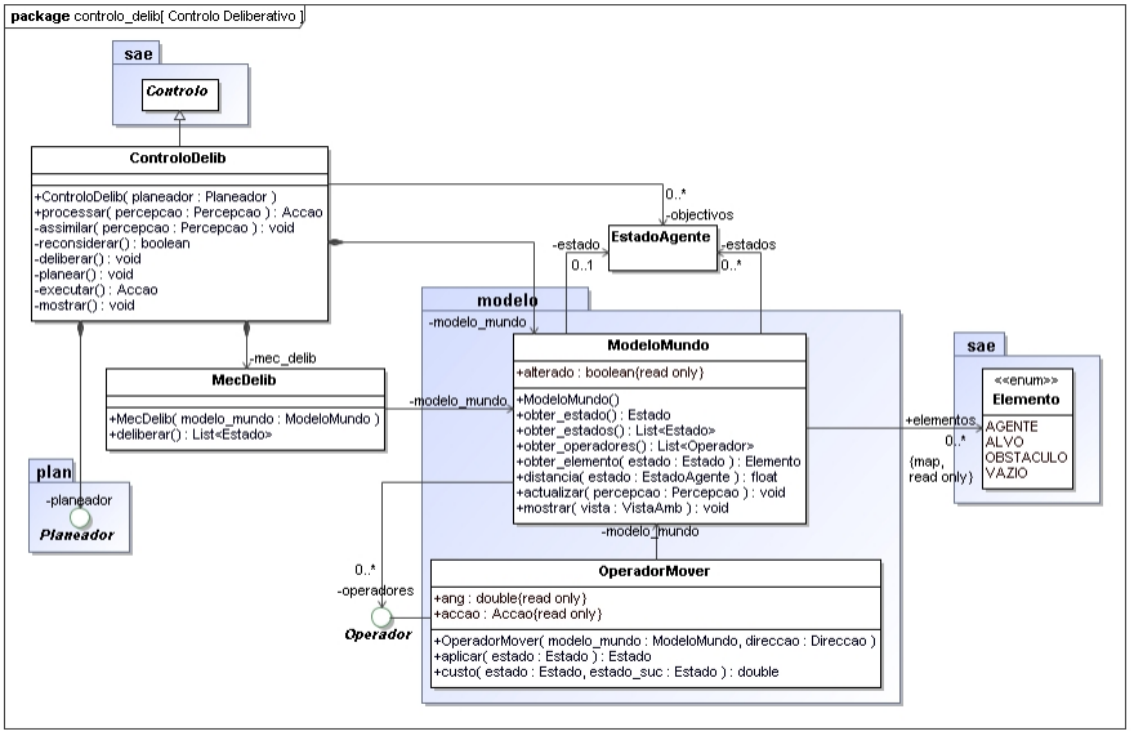


Figura 43 - Diagrama de classes do Controlo Deliberativo

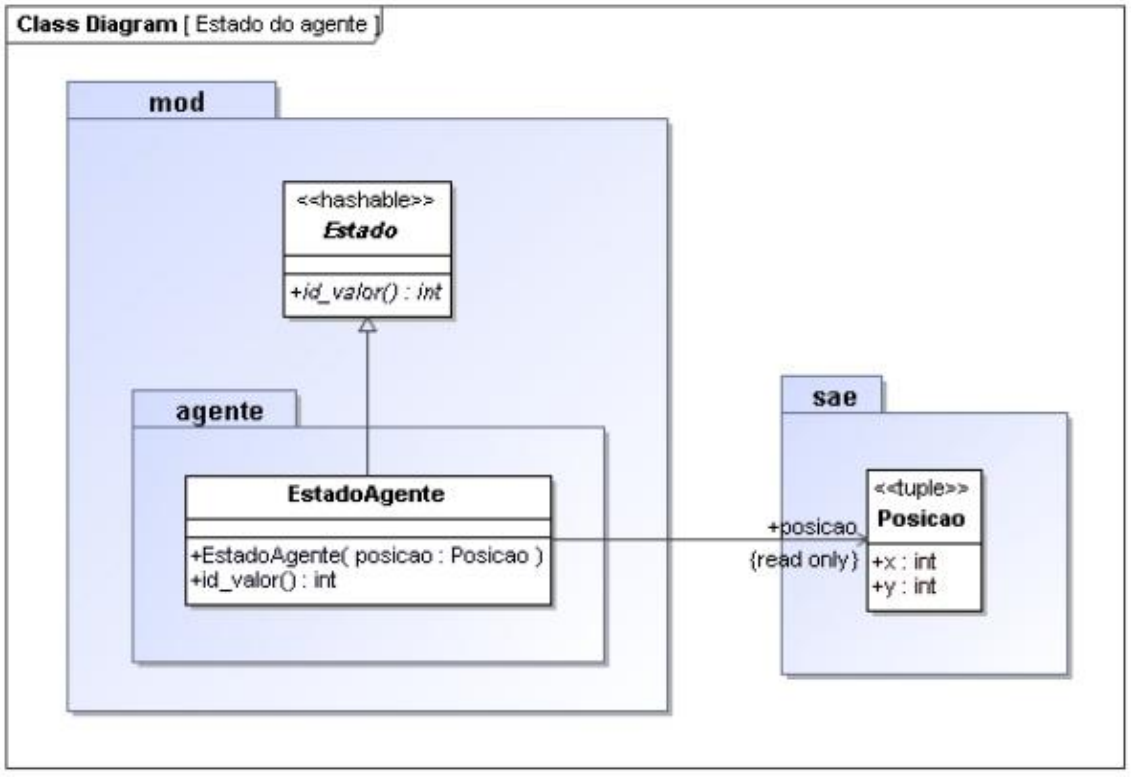


Figura 44 - Diagrama de classes do Estado do agente

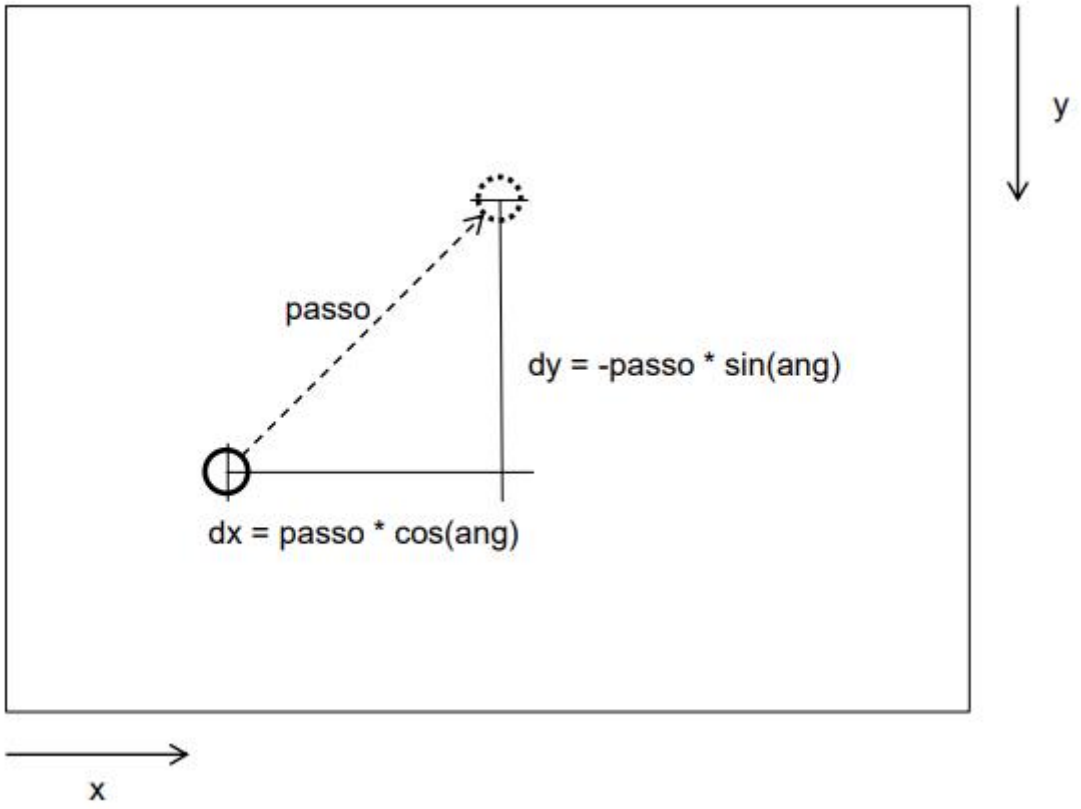


Figura 45 - Esquema da simulação do movimento por translação geométrica

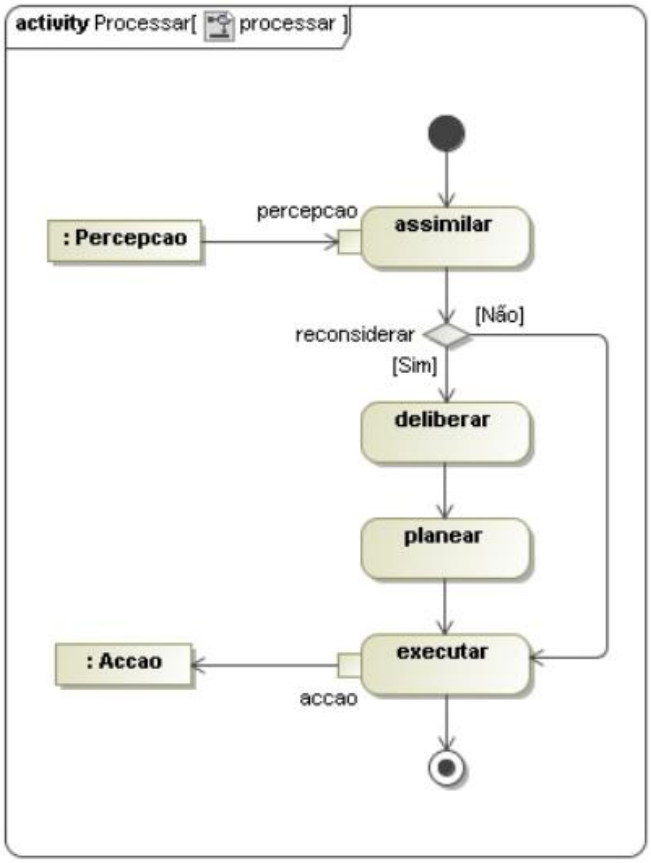


Figura 46 - Diagrama de atividade do método processar da classe *ControloDelib*

Planeamento Automático

Qualquer agente deliberativo precisa de fazer o planeamento das suas ações de maneira a chegar ao(s) objetivo(s).

Para isso, implementaram-se as bases para definir planos e planeadores, sendo estas as interfaces ***Planeador***, ***Plano*** e ***ModeloPlan***.

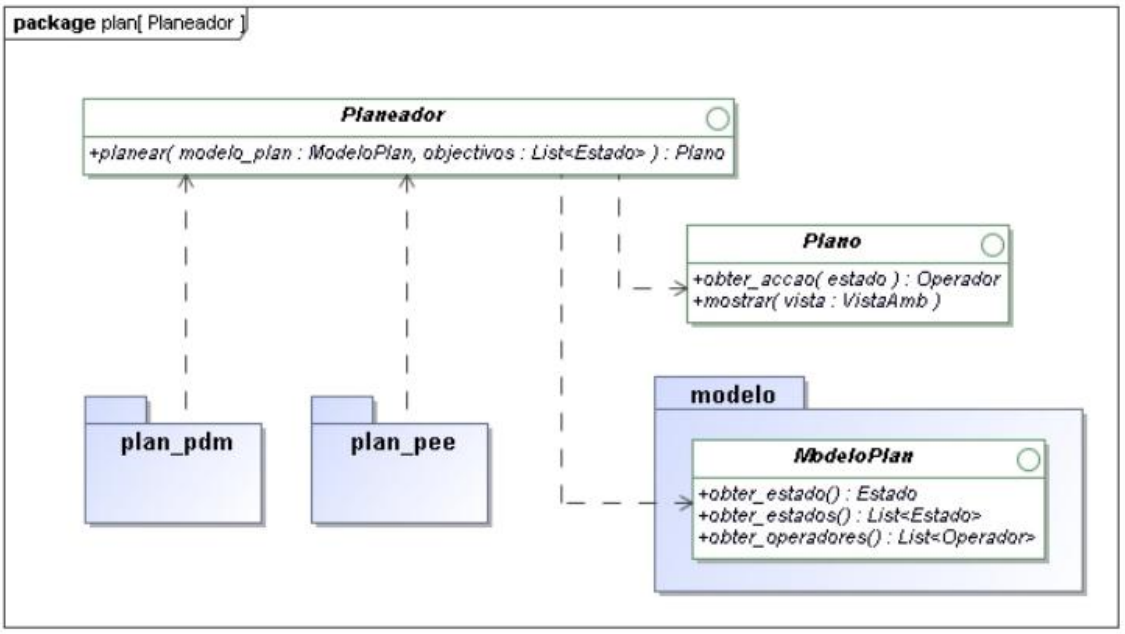


Figura 47 - Diagrama de classes da package plan

A interface ***Planeador*** define, de forma abstrata, o contrato para qualquer planeador em sistemas de agentes deliberativos, especificando que deve existir um método ***planear*** responsável por gerar planos (sequências de ações) para atingir objetivos definidos. Este recebe então um modelo do ambiente e uma lista de objetivos, explorando o espaço de estados através de algoritmos de procura automática (como PEE ou PDM) para encontrar a melhor sequência de ações que leva do estado inicial aos objetivos, retornando um objeto Plano ou *None* se não houver solução. Teoricamente, esta interface representa o componente de planeamento da arquitetura deliberativa, fundamental para que o agente decida "como fazer" para alcançar os fins definidos, baseando-se em modelos internos do ambiente e raciocínio automático.

A interface ***Plano*** estabelece um contrato abstrato que define o comportamento esperado de qualquer plano de ação em sistemas de agentes deliberativos. Esta interface modela um plano como uma sequência de ações destinada a conduzir o agente do estado atual até um estado objetivo. Para tal, inclui dois métodos abstratos essenciais: ***obter_accao(estado)***, que recebe o estado atual e devolve a próxima ação (ou operador) a executar, permitindo ao agente percorrer o plano de forma faseada; e ***mostrar(vista)***, que permite representar graficamente o plano, facilitando a sua análise e interpretação visual. Em termos concetuais, a interface ***Plano*** representa o produto do processo de planeamento — o raciocínio deliberativo sobre os meios para atingir um fim — sendo uma componente central para a execução sistemática e informada das ações do agente.

A interface ***ModeloPlan*** define o contrato abstrato que qualquer modelo de planeamento deve, especificando as componentes fundamentais que caracterizam um problema de planeamento. Em particular, esta interface fornece três métodos essenciais: ***obter_estado()***, que devolve o estado inicial do agente; ***obter_estados()***, que retorna o conjunto de todos os estados válidos no ambiente; e ***obter_operadores()***, que disponibiliza os operadores responsáveis pelas transições entre estados. Estes elementos permitem ao planeador compreender o ponto de partida, o espaço de estados e as ações possíveis, constituindo a base para a aplicação de algoritmos de procura e raciocínio automático. Em termos teóricos, a interface ***ModeloPlan*** assenta na necessidade de dispor de uma representação formal do problema para que o processo de planeamento seja viável, sendo uma componente imprescindível para que o agente consiga raciocinar sobre o ambiente, elaborar planos coerentes e tomar decisões informadas.

Planeador com base em PEE

O planeamento automático com base em Procura em Espaço de Estados (PEE) serve para gerar planos de ação para alcançar objetivos definidos. Utiliza um modelo de planeamento com estados, operadores e objetivos, representando o problema num espaço de estados. Através de mecanismos de procura, o planeador encontra uma sequência de ações que liga o estado inicial ao estado objetivo, sendo essencial em sistemas autónomos.

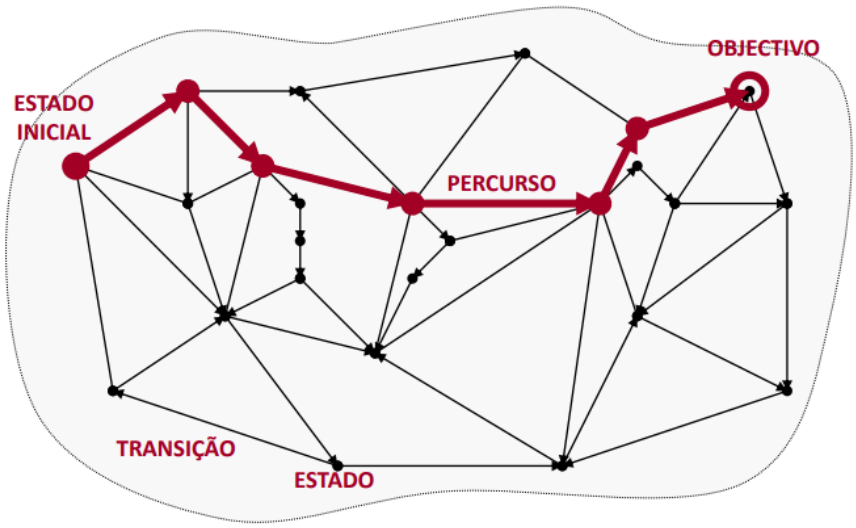


Figura 48 - Sequência de ações (percurso) do estado inicial até ao objetivo

De maneira a implementar este tipo de planeamento automático, construíram-se as seguintes classes: ***PlaneadorPEE***, ***PlanoPEE, ProblemaPlan*** e ***HeurDist***.

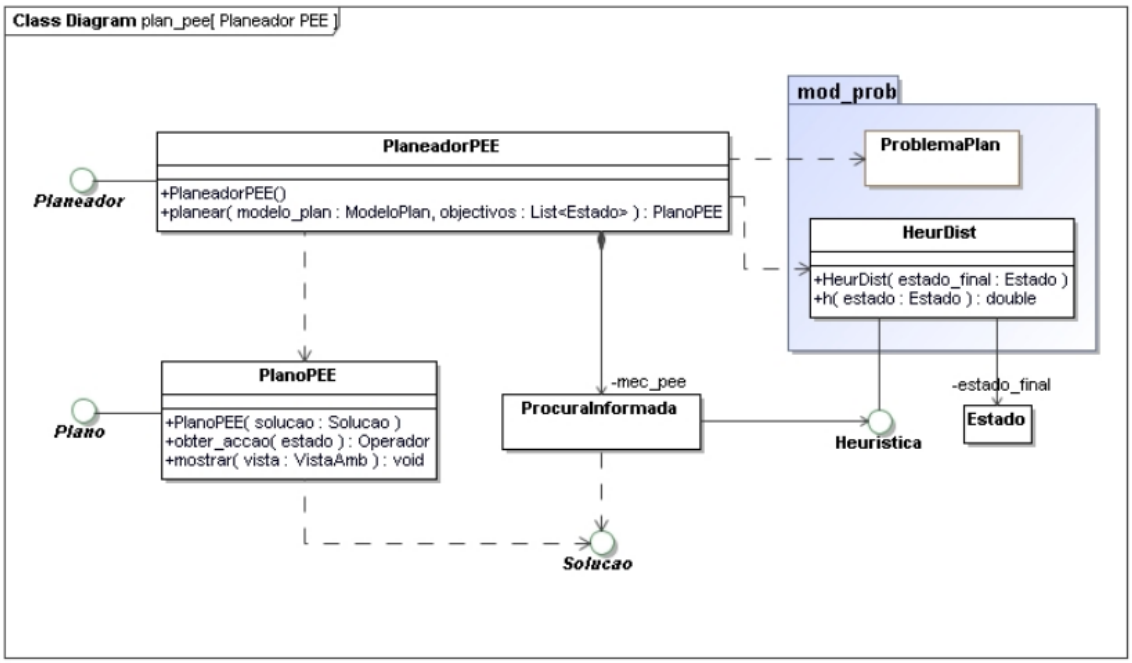


Figura 49 - Diagrama de classes da package plan_pee

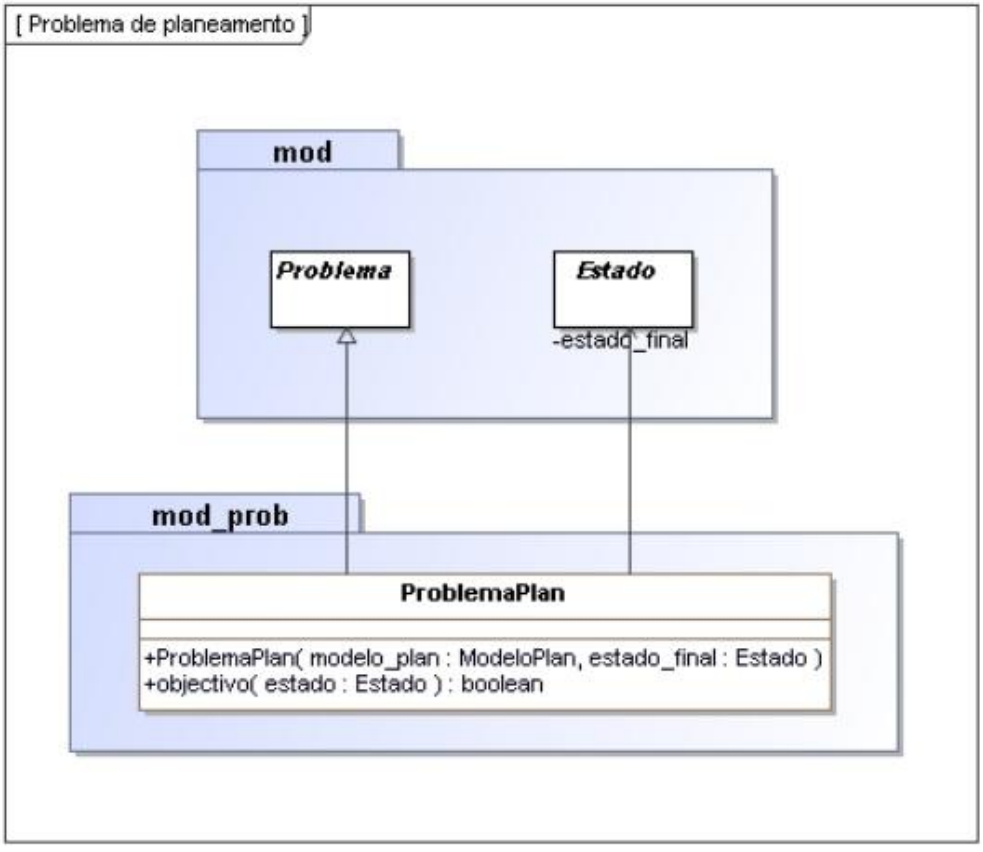


Figura 50 - Diagrama do Problema de planeamento

A classe ***PlaneadorPEE*** é uma implementação concreta da interface *Planeador* que utiliza o algoritmo de procura A* (através de *ProcuraAA*) para gerar planos ótimos em ambientes de agentes deliberativos. O seu método principal, ***planear(modelo_plam, objetivos)***, recebe um modelo de planeamento e uma lista de objetivos, seleciona o primeiro objetivo como estado final, constrói um problema de planeamento, define uma heurística de distância (*HeurDist*) e utiliza o mecanismo de procura A* para encontrar a solução mais eficiente. O resultado da procura é encapsulado num objeto *PlanoPEE*, que representa a sequência de ações a executar para atingir o objetivo. Esta classe concretiza o processo de planeamento automático baseado em procura informada, garantindo soluções ótimas quando a heurística é admissível, e integra-se na arquitetura deliberativa como a componente responsável por decidir "como fazer" para alcançar os fins definidos pelo agente.

Posteriormente, implementou-se a classe ***PlanoPEE***, que concretiza a interface ***Plano*** e oferece uma implementação específica para planos obtidos através de algoritmos de Procura em Espaço de Estados (PEE), como o A*. Esta classe estrutura internamente o plano como uma sequência ordenada de pares (estado, operador), que descrevem o trajeto desde o estado inicial até ao objetivo. O método ***obter_accao(estado)*** permite ao agente aceder à próxima ação prevista no plano, validando se o estado atual corresponde ao passo seguinte, o que assegura uma execução sequencial e coerente. Por sua vez, o método ***mostrar(vista)*** oferece uma representação gráfica do plano, através do desenho de vetores que ilustram as transições entre estados. Do ponto de vista teórico, ***PlanoPEE*** representa uma realização prática do conceito de plano como produto de um processo de procura, constituindo um mecanismo essencial para a execução estruturada e informada das ações de um agente deliberativo.

Depois, criou-se a classe ***ProblemaPlan***, uma especialização da classe ***Problema***, com o intuito de formalizar problemas de planeamento nos quais se pretende determinar uma sequência de ações que conduza o agente do estado inicial até um estado objetivo. Esta classe recebe, no seu construtor, um modelo de planeamento — responsável por fornecer o estado inicial e os operadores disponíveis — e o estado final, que representa o objetivo a atingir. Desta forma, são configurados o espaço de estados e as transições admissíveis. O método principal, ***objectivo(estado)***, permite verificar se um determinado estado corresponde ao estado objetivo, funcionando como critério de paragem para os algoritmos de procura. Em termos teóricos, ***ProblemaPlan*** constitui a formalização do planeamento como um problema de procura no espaço de estados, onde cada estado representa uma configuração possível do sistema, e as ações são modeladas como operadores. Esta abstração é essencial para a aplicação de técnicas de planeamento automático e para a operacionalização da arquitetura deliberativa dos agentes.

Por fim, implementou-se ***HeurDist***, que define uma heurística admissível baseada na distância euclidiana entre a posição atual de um estado e a posição objetivo, sendo utilizada em algoritmos de procura informada como o A*. O método principal ***h(estado)*** recebe um estado e calcula a distância linear até ao estado final, servindo como uma estimativa do custo mínimo necessário para atingir o objetivo, assumindo movimento

direto sem obstáculos. Esta abordagem garante admissibilidade, pois nunca sobrestima o custo real, sendo adequada para garantir a otimização de A*. Segundo a teoria, ***HeurDist*** fundamenta-se na necessidade de heurísticas em planeamento automático para guiar a procura de forma eficiente e é essencial para o funcionamento eficaz do ***PlaneadorPEE*** e de outros mecanismos de procura informada em agentes deliberativos.

Mecanismo PDM

O mecanismo de Processos de Decisão de Markov (PDM) é um método de planeamento automático para gerar planos em ambientes incertos. Baseia-se em modelos probabilísticos que representam estados, ações, transições e recompensas, permitindo a um agente autónomo tomar decisões que maximizem a utilidade esperada. Diferentemente da Procura em Espaço de Estados (PEE), o PDM lida com incertezas, modelando as probabilidades de transição entre estados e utilizando algoritmos como a programação dinâmica ou o aprendizado por reforço para encontrar políticas ótimas de ação.

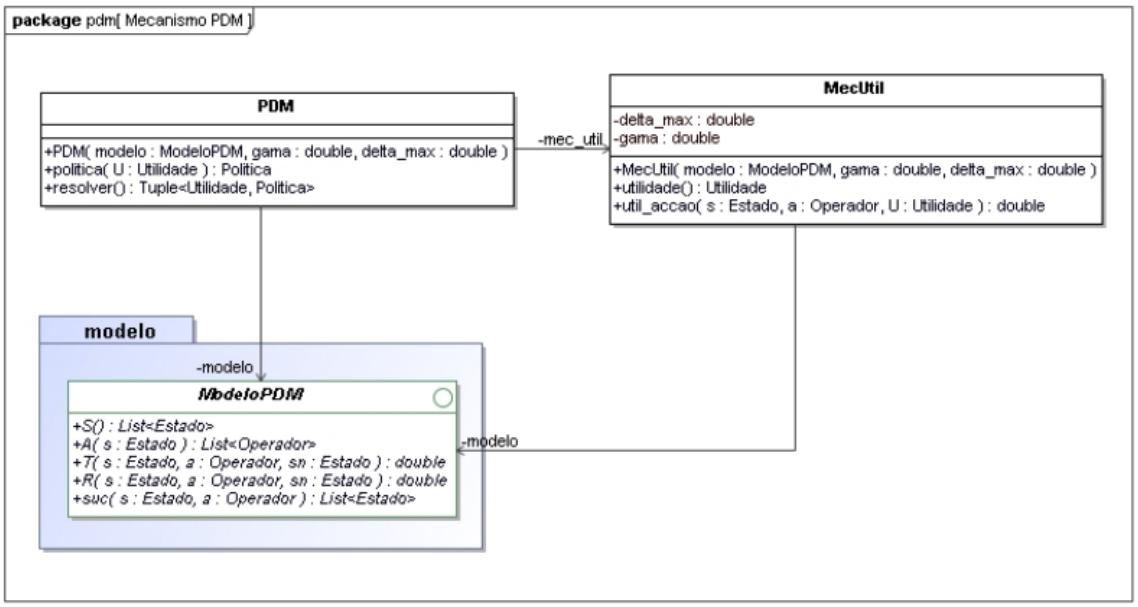


Figura 51 - Diagrama de classes da package pdm

Em primeiro lugar, definiu-se ***MecUtil***, que implementa o mecanismo de cálculo de utilidades ótimas para todos os estados num Processo de Decisão de Markov (PDM), recorrendo ao algoritmo de Iteração de Valor. O método principal, ***utilidade()***, inicializa as utilidades de todos os estados a zero e, de forma iterativa, atualiza-as usando a equação de Bellman até que a variação máxima entre iterações (***delta***) seja inferior a um limiar definido (***delta_max***), garantindo a convergência. Para cada estado, calcula a utilidade máxima considerando todas as ações possíveis, recorrendo ao método ***util_accao(s, a, U)***, que avalia a utilidade esperada de uma ação com base nas probabilidades de transição, recompensas e utilidades futuras descontadas pelo fator ***gama***. Teoricamente, esta classe concretiza o cálculo de utilidades globais para suportar a determinação da política ótima

em PDMs, sendo fundamental para agentes que tomam decisões racionais em ambientes estocásticos.

De seguida, concretizou-se a criação da classe PDM que implementa um Processo de Decisão de Markov, modelo matemático usado para representar decisões sequenciais em ambientes estocásticos, onde as transições entre estados são probabilísticas e recompensadas. No construtor, recebe um modelo de PDM (com estados, ações, transições e recompensas), um fator de desconto – ***gama*** – e um critério de paragem, ***delta_max***, inicializando o mecanismo de cálculo de utilidades (***MecUtil***). O método ***resolver()*** executa o algoritmo de iteração de valor para calcular as utilidades ótimas de cada estado e, a partir destas, deriva a política ótima, ou seja, a ação que maximiza a utilidade esperada em cada estado, recorrendo ao método ***politica()***. Pela teoria, esta classe concretiza a resolução ótima de problemas de decisão sequencial, aplicando as equações de Bellman e os princípios fundamentais dos PDMs, sendo essencial para agentes racionais que atuam em ambientes incertos.

Por fim, implementou-se a classe abstrata ***ModeloPDM***, que serve para modelar problemas PDM, especificando os métodos essenciais que qualquer implementação concreta deve fornecer: ***S()***, lista de todos os estados possíveis; ***A(s)***, ações disponíveis num dado estado; ***T(s, a, sn)***, probabilidade de transição entre estados dado uma ação; ***R(s, a, sn)***, recompensa associada a uma transição de estado via ação; e ***suc(s, a)***, estado sucessor resultante da aplicação de uma ação num estado. Estes métodos permitem descrever completamente a dinâmica, as possibilidades de ação e os critérios de avaliação de desempenho num ambiente estocástico, sendo indispensáveis para o cálculo de utilidades e políticas ótimas em PDMs.

Planeador com base em PDM

O planeador com base em Processos de Decisão de Markov (PDM) é um método de planeamento automático em inteligência artificial que opera em ambientes com incerteza. Utiliza um modelo probabilístico que define estados, ações, transições com probabilidades e recompensas, permitindo gerar políticas de ação que maximizam a utilidade esperada.

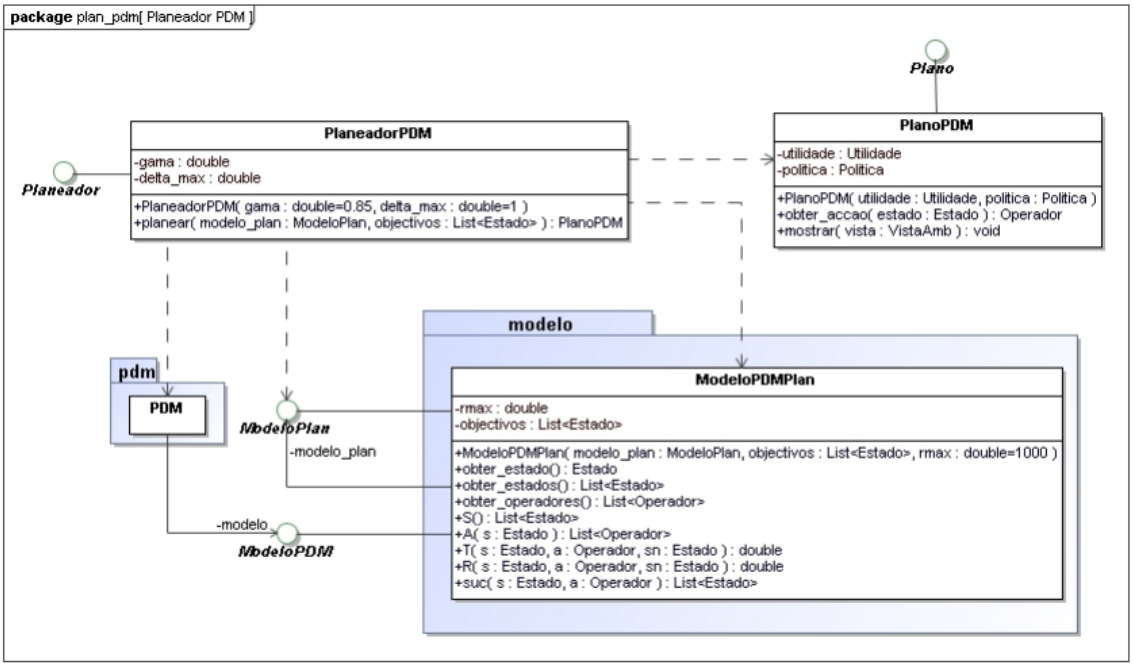


Figura 52 - Diagrama de classes da package plan_pdm

A classe ***PlaneadorPDM*** é uma implementação concreta da interface *Planeador* que utiliza PDM para gerar planos ótimos, baseando-se na maximização da utilidade esperada segundo a Equação de Bellman. O seu método principal, ***planear(modelo_plan, objectivos)***, recebe um modelo de planeamento e uma lista de objetivos, adapta o modelo para o formato de PDM (***ModeloPDMPlan***), instancia um objeto PDM com parâmetros de desconto (***gama***) e convergência (***delta_max***), e resolve o problema usando iteração de valor para calcular as utilidades e a política ótima. O resultado é encapsulado num objeto ***PlanoPDM***, que contém a utilidade e a política resultantes.

Já a classe ***PlanoPDM***, é uma implementação concreta da interface ***Plano***, concebida para representar planos baseados em PDM em sistemas autónomos. Armazena dois dicionários: um que associa cada estado à sua utilidade (valor esperado a longo prazo, calculado pela Equação de Bellman) e outro que mapeia cada estado à ação ótima a executar (política ótima). O método principal, ***obter_accao***, consulta a política para devolver a ação ótima correspondente ao estado atual, permitindo ao agente seguir a estratégia definida pelo planeador PDM; se o estado não estiver na política, retorna *None*. O método ***mostrar(vista)*** permite visualizar, numa interface gráfica, tanto os valores de utilidade de cada estado como os vetores de ação definidos pela política.

Finalmente, implementou-se a classe ***ModeloPDMPlan***, que integra e adapta as interfaces ***ModeloPlan*** e ***ModeloPDM*** para permitir o planeamento automático baseado em PDM. Esta classe recebe um modelo de planeamento genérico e uma lista de estados objetivo, construindo um modelo de PDM determinista onde as transições, ações e recompensas são definidas de acordo com o ambiente e os objetivos. Os métodos principais incluem ***S()***, ***A(s)***, ***T(s, a, sn)***, ***R(s, a, sn)*** e ***suc(s, a)***, que fornecem, respetivamente, o conjunto de estados, as ações possíveis num estado, a probabilidade de transição entre estados, a recompensa associada a cada transição e os estados

sucessores após uma ação. O método ***A*** garante que estados objetivo não têm ações associadas, enquanto ***T*** e ***suc*** usam um dicionário de transições deterministas para mapear pares (estado, ação) para o estado seguinte. O método ***R*** atribui uma recompensa máxima ao atingir um objetivo e recompensa nula nas restantes transições, refletindo a estrutura típica de problemas de planeamento com PDM.

Testes

O primeiro teste feito durante a implementação da parte 4, foi o ficheiro ***teste_delib.py***, utilizando o ***PlaneadorPEE***, onde se realizaram testes de integração ao ciclo deliberativo do agente, simulando o funcionamento dos principais módulos da arquitetura deliberativa. Este ficheiro inclui funções para obter perceções do ambiente (***obter_percecao***), atualizar o modelo do mundo com base nessas perceções (***actualizar_modelo_mundo***), gerar objetivos através do mecanismo de deliberação (***gerar_objectivos***) e criar planos de ação usando um planeador baseado em procura informada (***gerar_plano***). Cada função está documentada com exemplos de utilização (***doctest***), permitindo validar automaticamente se o agente consegue assimilar perceções, manter uma representação interna do ambiente, identificar objetivos relevantes e gerar planos para os atingir. Assim, ***teste_delib.py*** demonstra, na prática, a integração e funcionamento coordenado dos componentes de perceção, modelo do mundo, deliberação e planeamento, conforme indicado na teoria dos agentes deliberativos.

O último teste concretizado foi o ficheiro ***teste_pdm***, onde foi avaliado o funcionamento do mecanismo de PDM num ambiente linear determinístico 7x1, conforme descrito nos materiais teóricos. O ficheiro define a classe ***ModeloAmbiente7x1***, uma implementação concreta de ***ModeloPDM***, que modela sete estados dispostos linearmente, com dois estados terminais (1: perda, 7: ganho), ações de movimento à esquerda e à direita, transições determinísticas e recompensas associadas. Os métodos implementados (***S***, ***A***, ***T***, ***R***, ***suc***) fornecem ao ***PDM*** toda a estrutura necessária para calcular utilidades e políticas ótimas. No bloco principal, o teste cria uma instância do modelo e do ***PDM***, resolve o problema com fator de desconto 0.5 e imprime as utilidades e a política ótima resultantes. Os resultados mostram que as utilidades decrescem exponencialmente à medida que se afastam do estado objetivo, refletindo o impacto do fator de desconto, e que a política ótima consiste sempre em avançar para a direita, evitando ações que levam à perda. Este teste valida, na prática, a correta implementação dos conceitos teóricos de PDM, equações de Bellman e planeamento ótimo em ambientes determinísticos.

Agente Deliberativo e Agente Deliberativo PDM

A classe **AgenteDelib** implementa um agente autónomo com arquitetura deliberativa, integrando perceção, modelo do mundo, deliberação, planeamento e execução de ações. No construtor, inicializa o agente base e configura o controlo deliberativo, utilizando um planeador baseado em procura informada (***PlaneadorPEE*** que por sua vez utiliza a Procura A*) para gerar planos de ação orientados a objetivos. O método principal, ***executar***, realiza o ciclo completo de perceção-decisão-ação: obtém uma perceção do ambiente, processa essa perceção através do controlo deliberativo para gerar a próxima ação, atualiza a interface gráfica com o estado interno do agente e executa a ação no ambiente.

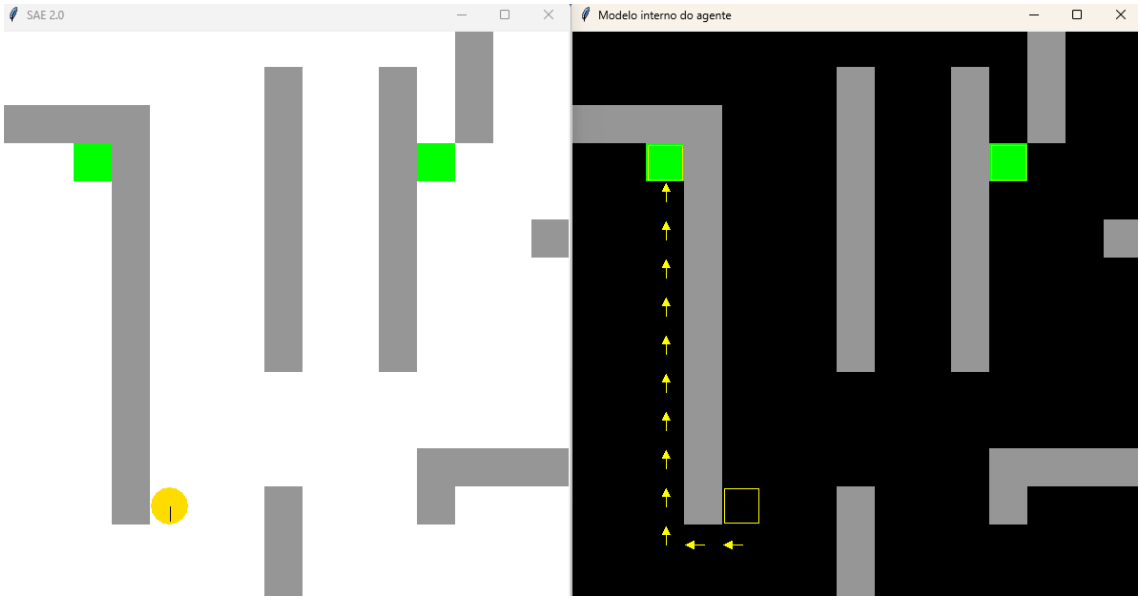


Figura 53 - Execução do agente deliberativo no ambiente 3 do SAE, utilizando o PlaneadorPEE | Primeiro objetivo

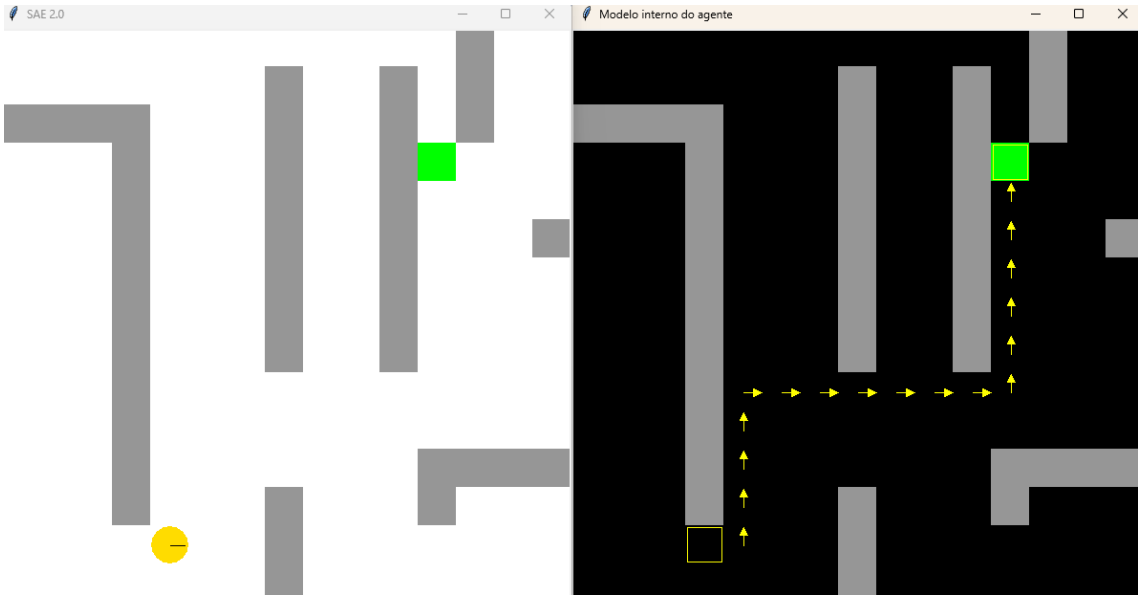


Figura 54 - Execução do agente deliberativo no ambiente 3 do SAE, utilizando o PlaneadorPEE | Segundo objetivo

Enquanto o ***AgenteDelib*** usa como controlo deliberativo um planeador de procura em espaço de estados, o ***AgenteDelibPDM*** usa o planeador de PDM, para tomar decisões

racionais em ambientes incertos, integrando perceção, controlo deliberativo e planeamento baseado em políticas ótimas. No construtor, inicializa o agente base e configura o controlo deliberativo com um planeador PDM (***PlaneadorPDM***), como já foi dito, ajustando o fator de desconto (***gama***) para valorizar recompensas futuras e garantir a convergência do cálculo de utilidades. O método principal, ***executar***, realiza o ciclo deliberativo clássico: obtém a perceção do ambiente, processa essa perceção para determinar a ação ótima segundo a política calculada pelo ***PDM***, exibe o plano (utilidade e política) na interface gráfica e executa a ação no ambiente. Definiu-se ***gama*** como 0.95 para dar mais peso às recompensas futuras, incentivando o agente a considerar o impacto a longo prazo das suas ações. Isto é crucial para alcançar objetivos distantes ou múltiplos, especialmente em ambientes grandes.

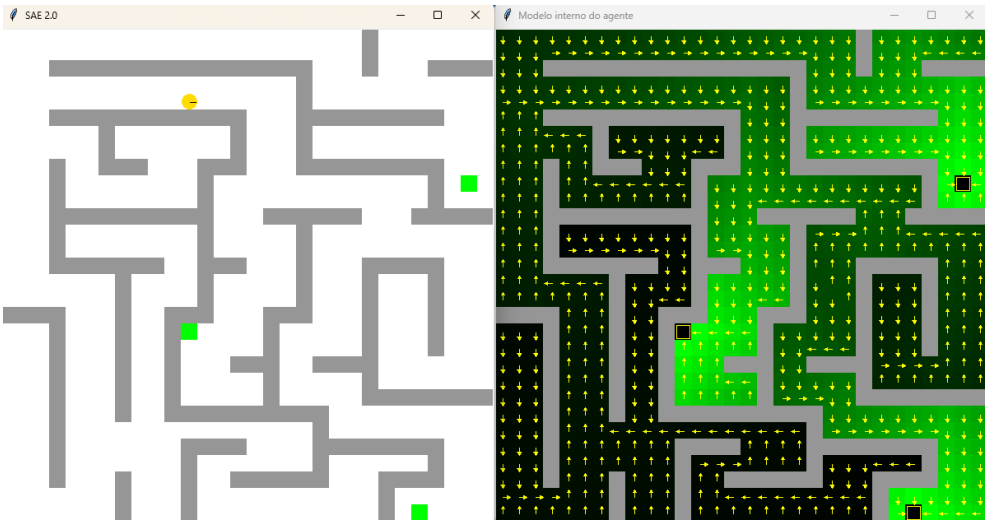


Figura 55 - Execução do AgenteDelibPDM num ambiente grande | Primeiro objetivo

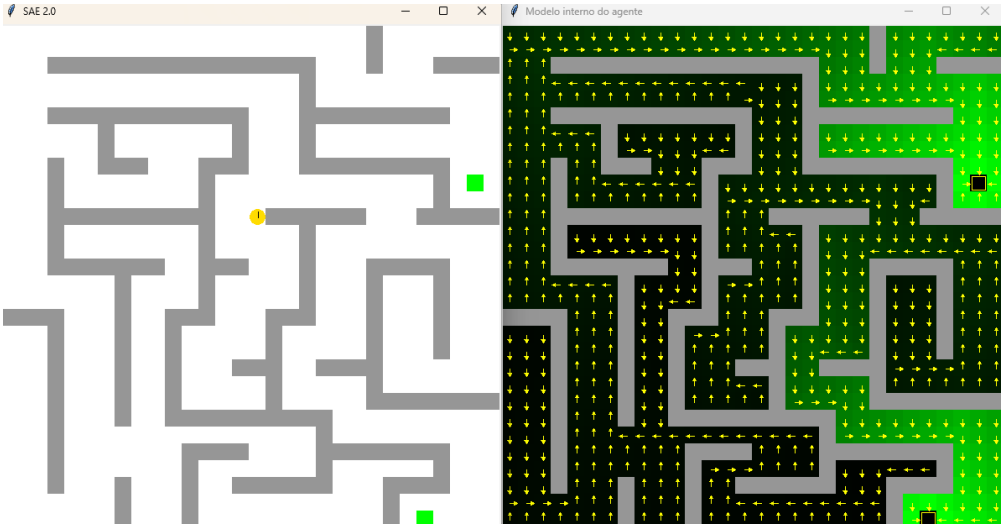


Figura 56 - Execução do AgenteDelibPDM num ambiente grande | Segundo objetivo

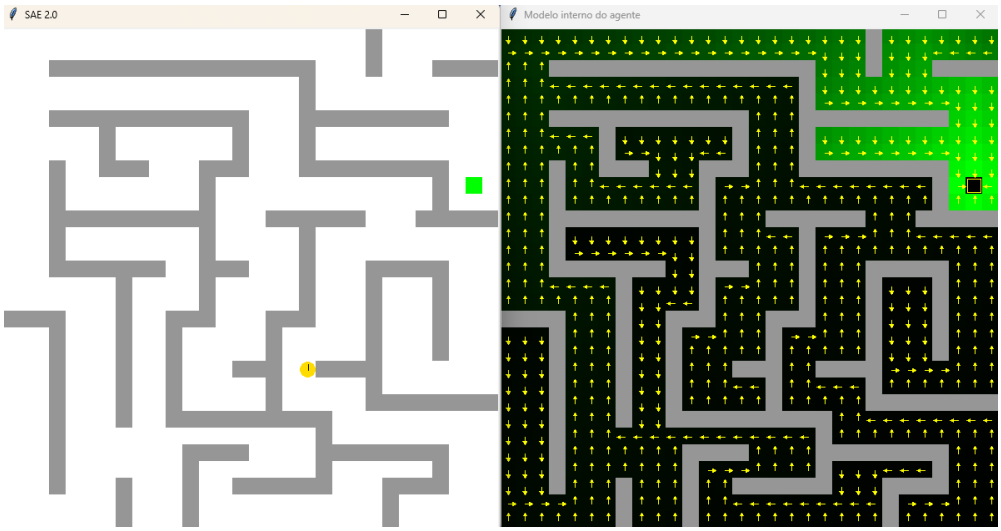


Figura 57 - Execução do AgenteDelibPDM num ambiente grande | Terceiro objetivo

Nota: não se deve usar ***gama*** = 1 devido ao facto de as recompensas futuras não serem descontadas, ou seja, têm o mesmo peso que as recompensas imediatas, tratando todas as recompensas ao longo do tempo como iguais. Isto leva a um cálculo infinito, pois a soma das recompensas futuras não converge, e assim o agente não consegue determinar uma política ótima, resultando num comportamento indefinido/indeciso.

Exercício final proposto

Foi pedido para que o agente voltasse ao estado inicial nos dois tipos métodos de procura: PEE e PDM.

A implementação feita foi em ***ModeloMundo*** inicializando no construtor um atributo, ***estado_inicial***, para guardar o estado inicial do agente, que inicialmente está vazio. Após criar este atributo, concretizou-se, no método ***actualizar(percepcao)***, uma verificação no início do método onde se averigua se o estado inicial é *None*, e caso seja, é definido ***estado_inicial*** como a posição atual do agente. Esta implementação foi aqui feita devido ao facto de este método atualizar o modelo do mundo com base nas novas informações percebidas pelo agente.

De seguida, implementou-se uma verificação no método ***__planear()*** para, caso não haja objetivos, o plano de ação gerado é um plano cuja única ação é voltar ao estado inicial. A razão para isto ter sido feito neste método é que este é responsável por gerar um plano de ação, ou seja, o que o agente deve fazer a seguir.

Apesar de esta concretização funcionar e após a explicação a do docente, percebeu-se que a implementação feita em ***__planear()***, devia ter sido feita em ***__deliberar()***, pois é esta função que permite ao agente perceber o que vai fazer, e é necessário o mesmo raciocinar sobre os fins antes de executar o plano de ação. Uma correção funcional a ser feita é, por exemplo, acrescentar uma verificação em ***__deliberar()*** para averiguar se a lista de objetivos está vazia ou não, e caso esteja, atribui-se à lista de objetivos um único objetivo, que é o estado inicial do agente.

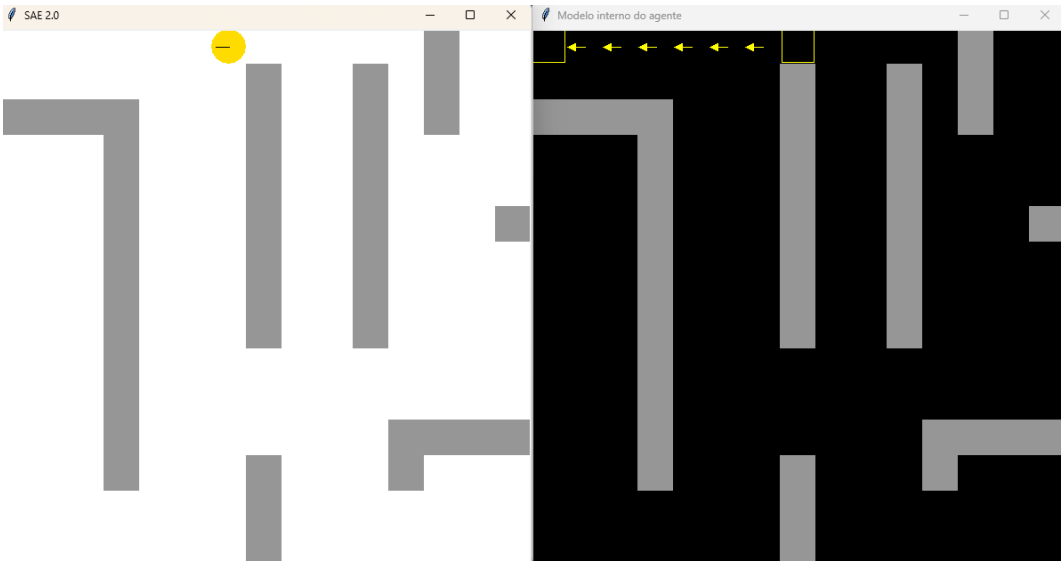


Figura 58 - Execução do agente deliberativo no ambiente 3 do SAE, utilizando o PlaneadorPEE | Estado Inicial

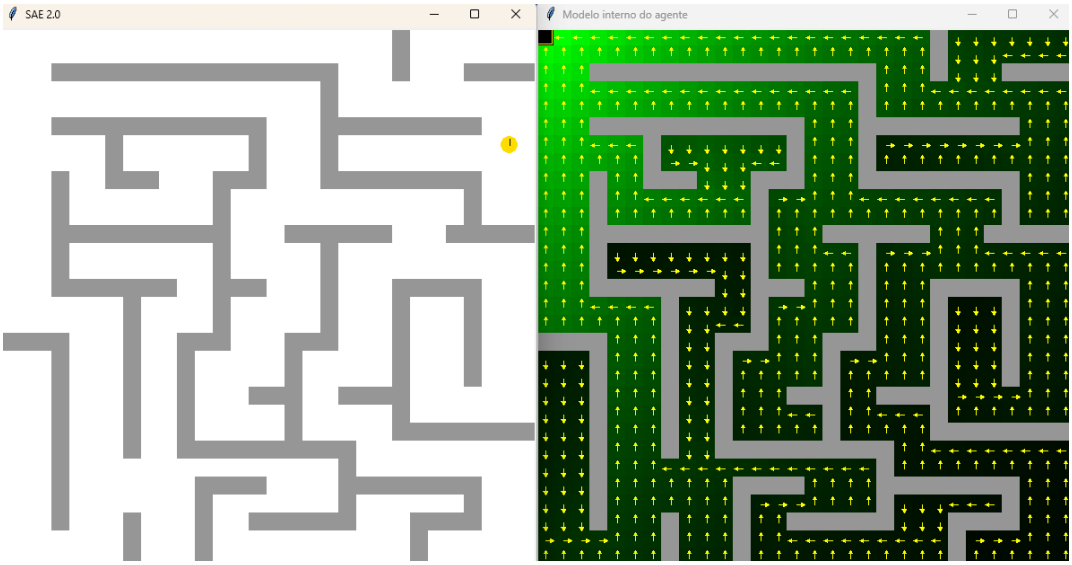


Figura 59 - Execução do AgenteDelibPDM num ambiente grande | Estado Inicial

Revisão do projeto

Da entrega 10 para a 11:

- ***OperadorMover*** não tinha os atributos ***ang*** e ***accao*** e as suas respetivas *properties*, pois são *read-only*;
- Na classe ProcuraMelhorPrim, foi eliminado o ***num_estados_repetidos***, pois não fez sentido consoante o que foi pedido pelo professor;
- Em ***EstadoAgente***, adicionou-se a *property* ***posição***, pois no diagrama mostra *read-only*;
- Houve um erro feito ao contar os nós repetidos, na classe ***Contagem***, na parte da solução, não foi feito ***mec_proc.nos_repetidos***, e sim ***mec_proc.nos_estados_repetidos***, que foi a variável usada inicialmente antes de alterar o raciocínio;

Da entrega 11 para a 12:

- ***ModeloMundo***: no método ***actualizar()***, o ***if*** tinha ***self.alterados*** em vez de ***self.alterado***;
- ***MecDelib***: na função ***__selecionar_objetivos(objetivos)***, removi os parênteses à frente de ***self.__modelo_mundo.distancia*** pois ***distancia*** é um atributo ;
- ***ControloDelib***: em ***__planear()***, faltou chamar a função ***planear*** no ***self.__planeador***, de modo que o agente possa de facto gerar um plano de ação para alcançar os objetivos pretendidos;
- Ainda em ***ModeloMundo***: faltou criar a propriedade *getter* do ***self.__alterado***, pois este atributo é apenas de leitura;
- ***OperadorMover***: alterou-se ***self.ang*** e ***self.accao*** para ***self.__ang*** e ***self.__accao***, pois estes atributos são privados e não públicos. No construtor, em ***self.__ang*** e ***self.__accao***, estava ***self.__ang = direccao.value*** e ***self.__accao = Accao(direccao)***, e foi corrigido para ***self.__ang = self.__direccao.value*** e ***self.__accao = Accao(self.__direccao)***, pois não se estava a usar corretamente o atributo ***self.__direccao***

Conclusão

Este projeto proporcionou uma visão abrangente e prática sobre o desenvolvimento de agentes inteligentes capazes de operar de forma autónoma em ambientes dinâmicos. Ao longo das quatro partes, foram explorados conceitos fundamentais, desde a arquitetura básica de sistemas autónomos até técnicas avançadas de planeamento e controlo, passando por comportamentos reativos e deliberativos.

Os principais conhecimentos adquiridos incluem a modelação de ambientes e agentes, a implementação de comportamentos reativos baseados em estímulos e respostas, e a aplicação de algoritmos de procura para resolução de problemas. Além disso, o projeto

permitiu compreender a importância do planeamento automático, utilizando métodos como PEE (Procura em Espaço de Estados) e PDM (Processos de Decisão de Markov), que conferem ao agente a capacidade de tomar decisões ótimas em cenários complexos.

Em resumo, este trabalho consolidou competências essenciais no domínio da IA, destacando-se a integração de diferentes níveis de controlo (reativo e deliberativo) e a aplicação de técnicas de procura e planeamento para criar sistemas autónomos eficientes e adaptativos. O projeto não só reforçou a compreensão teórica, mas também desenvolveu habilidades práticas na implementação de agentes inteligentes, preparando para desafios futuros em sistemas autónomos e robótica inteligente.

Bibliografia/Webgrafia

- Figuras dos diagramas retiradas dos *pdfs* de implementação do projeto, escritos por Luís Morgado;
- Teoria e implementação obtidas a partir dos *pdfs* escritos pelo docente e a partir das aulas;
- Sommerville, Ian – *Software Engineering: 10th Edition*. England: *Pearson Education Limited*, 2016. ISBN 978-0-13-394303-0;
- *Geeks for Geeks*, *Class Diagram | Unified Modeling Language* (UML), disponível em: https://www.geeksforgeeks.org/unified-modeling-language-uml-class-diagrams/;
- *A* Algorithm | Solved Example | Informed Search | Artificial Intelligence*, disponível em: https://www.youtube.com/watch?v=vsqndiGTD8k;
- *Greedy Best First Search Algorithm | Informed Search Algorithm | Artificial Intelligence*; disponível em: https://www.youtube.com/watch?v=MXBMXhPzw9I;
- *Heuristic Functions in Informed Search Algorithms | Artificial Intelligence,* disponível em: https://www.youtube.com/watch?v=WM0pxFENw58;
- *Iterative Deepening Depth First Search | Uninformed Search | Artificial Intelligence,* disponível em: https://www.youtube.com/watch?v=nY3okMsCIlY;
- *Depth Limited Search Algorithm in Artificial Intelligence | Uninformed Search Algorithm,* disponível em: https://www.youtube.com/watch?v=-bD0kiDWjIQ;
- *Depth First Search in Artificial Intelligence | DFS in AI | Uninformed Search,* disponível em: https://www.youtube.com/watch?v=1Y4cMGH7c14;
- *Uniform Cost Search Algorithm in Artificial Intelligence | Uninformed Search Strategy,* disponível em: https://www.youtube.com/watch?v=_ASAGjvI980;
- *Breadth First Search (BFS) in Artificial Intelligence | Uninformed Search Algorithm,* disponível em: https://www.youtube.com/watch?v=ksinr15Nlmo;
- *Intelligent Agents in Artificial Intelligence,* disponível em: https://www.youtube.com/watch?v=oCQchV3X6wg;