INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA
Departamento de Engenharia Eletrónica e Telecomunicações e Computadores
Licenciatura em Engenharia Informática e Multimédia

Relatório Final

Diogo Lobo
48168 TLEIM41D

Inteligência Artificial para Sistemas Autónomos

Professor:
Eng. Luís Morgado

07/2023

Conteúdo

Lista de Figuras

1 Introdução

O trabalho continuo desenvolvido nesta cadeira tem como objetivo o desenvolvimento de arquiteturas de agentes capazes de serem utilizados em vários e diferentes sistemas autónomos e a aprendizagem da utilidade e como usar os diferentes diagramas UML existentes.

Este projeto enquadra-se nas áreas de inteligência artificial e engenharia de software e para a realização deste projeto é necessário consolidar e saber aplicar a matéria lecionada durante as aulas sobre os diferentes tipos de arquitetura e o uso de UMLs.

Para ser possível alcançar o objetivo deste projeto foram utilizadas duas linguagens de programação diferentes, Java e Python no editor de código "Visual Studio Code".

2 Enquadramento Teórico

Esta cadeira enquadra-se nas áreas de Inteligência Artificial e Engenharia de Software e está organizada em 7 diferentes temas dentro destas áreas.

1. Introdução à Inteligência Artificial
2. Introdução à Engenharia de Software
3. Arquitetura de Agentes Inteligentes
4. Arquitetura de Agentes Reativos
5. Raciocínio Automático e Tomada de Decisão
6. Arquitetura de Agentes Deliberativos
7. Aprendizagem por Reforço

2.1 Parte 1- Introdução à Inteligência Artificial

Inteligência artificial é um ramo das ciências da informática que estuda o desenvolvimento de sistemas computacionais capazes de obterem um comportamento inteligente.
Este ramo adota também duas principais perspetivas, **perspetiva analítica** e **perspetiva sintética**.

Perspetiva Analítica Perspetiva que, por via empírica, é suposta desenvolver modelos e teorias para explicar os fenómenos observados e reproduzir esses fenómenos em sistemas artificiais.

Perspetiva Sintética Perspetiva que tendo em base modelos e teorias realizados pela perspetiva analítica, é suposta desenvolver sistemas capazes de apresentar características e comportamentos associados ao conceito de inteligência.

A inteligência artificial tem três principais conjuntos de ideias, ou paradigmas, que servem de base e orientam a maneira de abordar as questões e problemas de uma determinada área e de idealizar as respetivas soluções para essas questões.
Esses paradigmas são:

- Paradigma simbólico
- Paradigma Conexionista
- Paradigma Comportamental

Paradigma Simbólico A inteligência é resultante da ação de processos computacionais sobre estruturas simbólicas.
Neste paradigma, os simbolos são centrais no processamento de informação e na representação de conceitos relacionados com o dominio de um problema sob a forma de estruturas simbólicas.

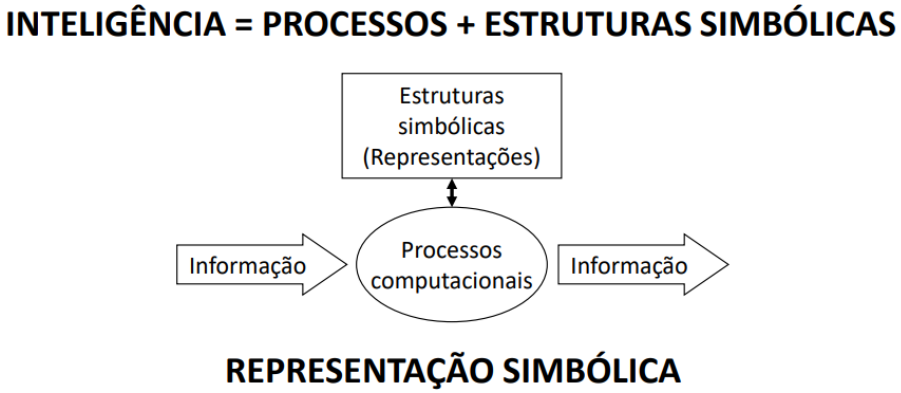


Figura 1: Processo do paradigma simbólico

As representações simbólicas não têm significado intrínseco, o significado resulta das relações entre as estruturas de tais símbolos.

Paradigma Conexionista A inteligência é uma propriedade emergente das interações de um número elevado de unidades elementares de processamento interligadas entre si.
Neste paradigma, o processamento de informação é baseado em redes de unidades de processamento elementares (designadas por neurónios) interligadas entre si, sendo a informação e processada nessas redes de forma distribuída, em particular nas ligações entre neurónios, em vez de símbolos.

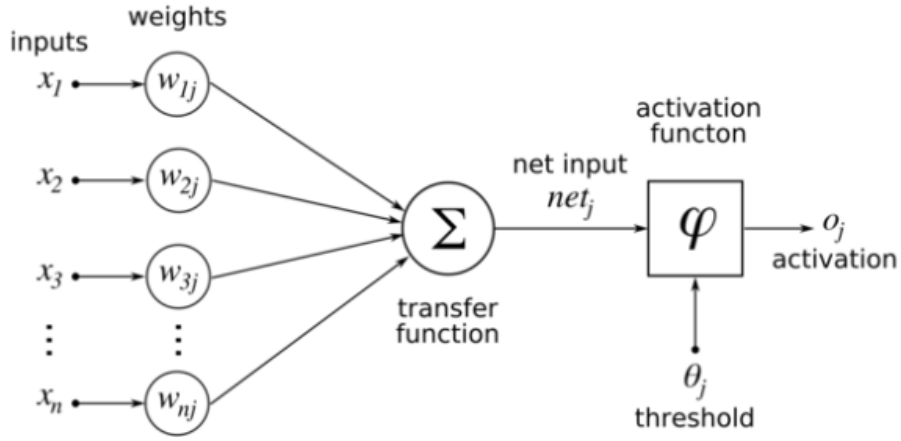


Figura 2: Processo do paradigma Conexionista

Paradigma Comportamental A inteligência resulta do comportamento individual e conjunto de múltiplos sistemas a diferentes escalas de organização tendo por base relações entre estímulos e respostas.
Neste paradigma, o processamento de informação é baseado em relações entre estímulos e respostas, modeladas sob a forma de reações e comportamentos(conjuntos estruturados de reações).

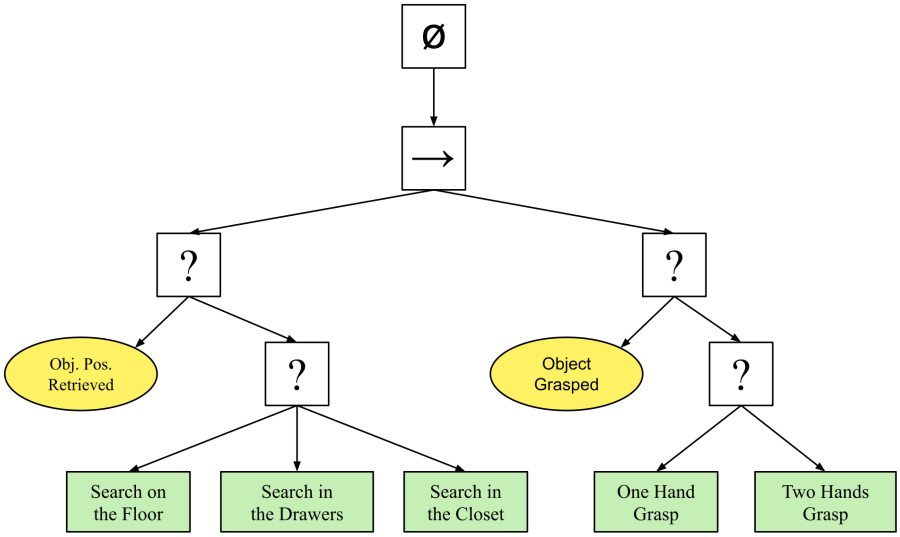


Figura 3: Exemplo de árvore comportamental

Dos ramos existentes da área da inteligência artificial este semestre foi mais focado num ramo em particular, sendo esse ramo a inteligência artificial para sistemas autónomos.

Definição de Autónomo: Do grego autónomos, que se governa por leis próprias; independente; autossuficiente; sistema que funciona sem depender de ligação a outro sistema
Os sistemas autónomos inteligentes operam num ciclo de **perceção-processamento-ação** onde é realizado o controlo da função de sistema de modo a concretizar a finalidade desse sistema.

2.2 Introdução à Engenharia de Software

A engenharia de software é uma área da engenharia informática orientada para a especificação, desenvolvimento e manutenção de software de modo sistemático e quantificável.

Sistemático Capacidade de realizar o desenvolvimento de software de forma organizada e previsível, de modo a garantir a satisfação dos requisitos definidos, incluindo tempo e recursos necessários.

Quantificável Capacidade de avaliar os meios envolvidos e os resultados produzidos no desenvolvimento do software, utilizando métodos e processos adequados para garantir a qualidade e o desempenho do software produzido, bem como a criação de documentação e a monitorização do processo de desenvolvimento.

Com o crescimento da utilização de sistemas baseados em software, que vão desde aplicações informáticas e sistemas de informação a sistemas de controlo industrial ou de veículos autónomos. Com esta evolução viram-se dois problemas a ganharem mais relevantes sendo esses a **Complexidade** e **Mudança**.

Na engenharia de software é utilizado regularmente uma linguagem chamada de UML (Unified Modeling Language), uma linguagem de modelação geral de sistemas, orientada em particular para a modelação de software.

Vantagens

- Orientada para concepção e modelação de sistemas de forma organizada e sistemática.
- Linguagem normalizada que facilita a compreensão e comunicação dos modelos.
- Facilitação da produção de código
- Adopção de uma abordagem orientada a objetos.

Esta linguagem está dividida em classificadores (tipos de elementos base), diagramas, mecanismos de extensão e complementos de informação. Esta linguagem utiliza diagramas, cada um com o seu propósito único para representar diferentes aspetos de um sistema.

Tipos de Diagramas

- Diagramas de Classe
- Diagramas de sequência
- Diagramas de Atividade
- Diagramas de Transição de Estado
- Diagramas de Módulos (Packages)

Esta linguagem vai ser muito útil no desenvolvimento do projeto pela simplificação das classes e a interação entre cada uma delas.

2.3 Arquitetura de Agentes Inteligentes

Os sistemas autónomos utilizam algo chamado por Agente Inteligente, uma representação computacional do sistema autónomo inteligente. Este agente realiza os três passos do ciclo do sistema autónomo.

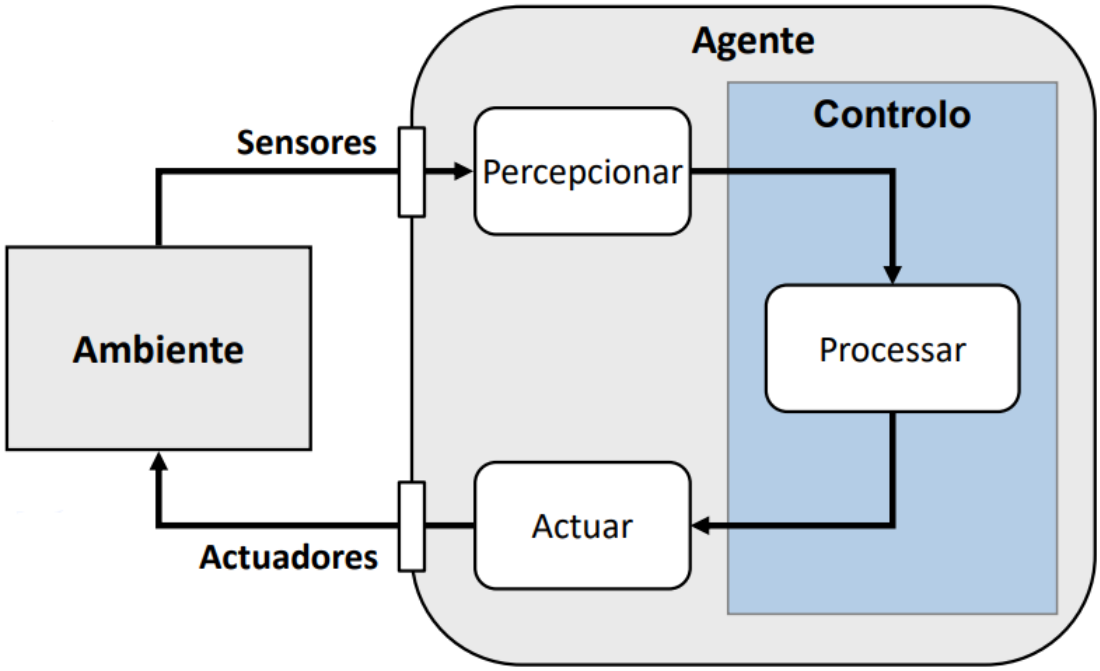


Figura 4: Ciclo do agente inteligente

Este agente inteligente atua num ambiente, que pode ser tanto virtual(por exemplo um npc de um jogo) como num ambiente real(por exemplo um robô de limpeza) e esta representação de ambiente é um elemento de suporte do processamento interno de alguns agentes inteligentes.
Essas representações podem ter diferentes níveis de complexidade (por exemplo, representações com base em grafos). No caso de um ambiente físico contínuo requer a discretização das perceções. São utilizados sensores para o agente percecionar qual o seu próximo "movimento"e utiliza atuadores para realizar a ação de acordo com esse movimento.
Um agente inteligente tem 4 características principais:

- Autonomia
- Reatividade
- Pro-Atividade
- Sociabilidade

que em conjunto estão associadas a uma finalidade do agente, ou seja, o seu propósito.

Esta inteligência tem duas vertentes principais ao seu conceito, **Cognição** e **Racionalidade**.

Cognição O processo de conhecer, podendo ser também caracterizada como uma propriedade global do agente, expressa através da capacidade de realizar a ação adequada dadas as condições do ambiente.

Racionalidade A capacidade de decidir no sentido de conseguir o melhor resultado possível perante os objetivos atribuídos.
Um sistema é racional se faz a "ação certa"dado o conhecimento que possui. [Russell e Norvig, 2003]

2.4 Arquitetura de Agentes Reativos

Neste tipo de arquitetura o comportamento do sistema é gerado de forma reativa, com base em associações entre estímulos e respostas.
Os objetivos são implícitos nas associações estímulos-resposta que definem o comportamento do agente. Existe um acoplamento forte com o ambiente, em alguns casos direto, através de associações entre os estímulos derivados das perceções e as respostas que produzem ações.

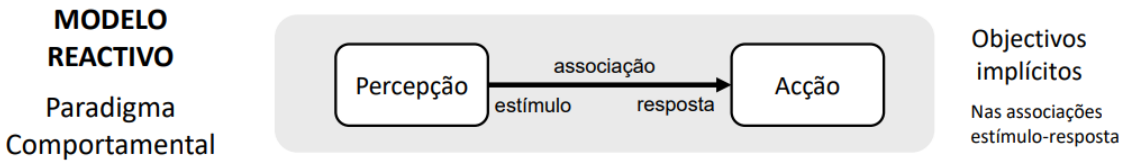


Figura 5: Modelo Reativo

Veículos de Braitenberg Um exemplo deste tipo de arquitetura é os veículos de Braitenberg, dispositivos robóticos simples, para estudar a forma como comportamentos complexos podem resultar de mecanismos reativos relativamente simples.

Uma arquitetura de agentes reativos define um ciclo perceção-reação-ação, onde as reações definem de forma modular as associações entre estímulos(derivados das perceções) e respostas(geradoras da ação).

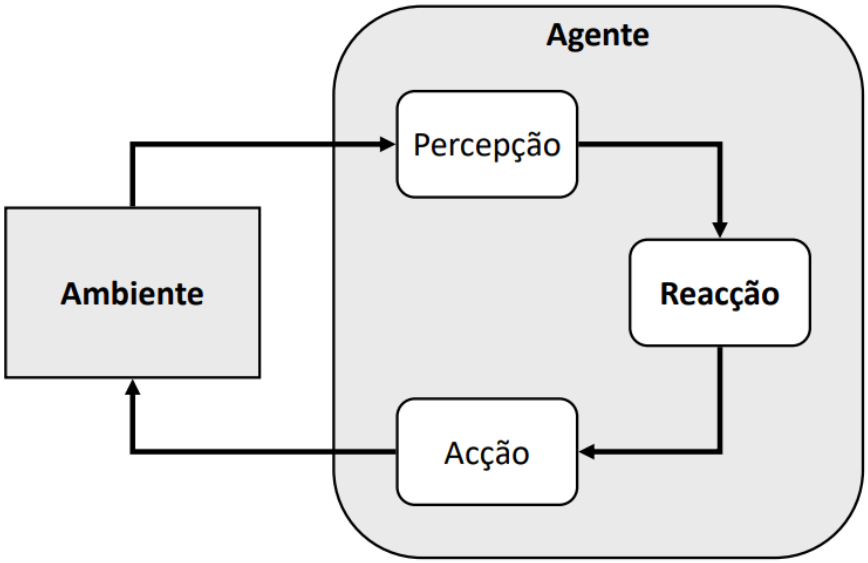


Figura 6: Arquitetura de um agente reativo

Estímulo Define informação ativadora de uma reação

Resposta Define uma resposta a estímulos, em termos de ação a realizar e da respetiva prioridade

Reação Módulo que associa estímulos a respostas.

Um agente reativo é composto por um conjunto de reações, as quais devem ser organizadas em módulos comportamentais designados por **comportamentos**, de modo a reduzir a complexidade e a facilitar o desenvolvimento e manutenção do agente.

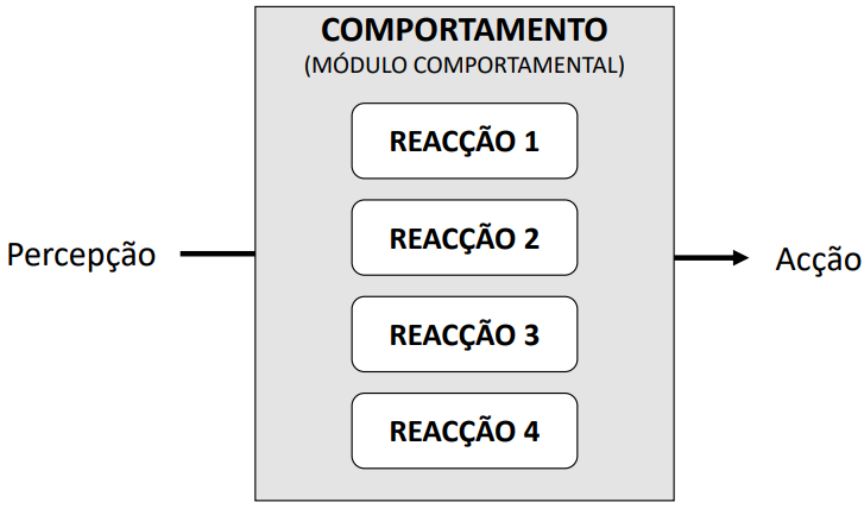


Figura 7: Exemplo de módulo comportamental

Um comportamento realiza de forma modular e coesa o encapsulamento das reações internas, contribuindo assim para reduzir o acoplamento e a complexidade da arquitetura de um agente reativo, relaciona padrões de perceção com padrões de ação e no caso geral um comportamento pode ser composto por outros comportamentos, sendo designado de **Comportamento Composto**.

Comportamento Composto

- Agrega conjuntos de comportamentos
- Requer um mecanismo de seleção de ação para determinar a ação a realizar em função das respostas dos vários comportamento internos

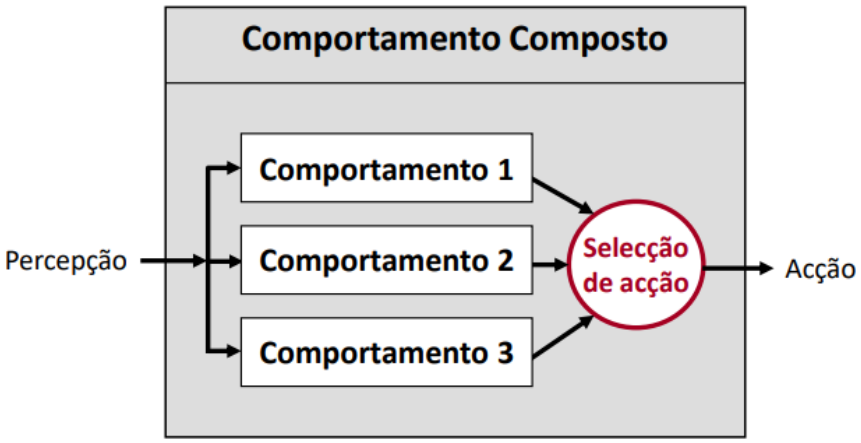


Figura 8: Exemplo de um comportamento composto

Seleção de Ação

- Execução paralela de ações
- Combinação de ações
- Precedência de ações

Na combinação de ações existem 3 possíveis coordenações desses comportamentos.

Hierarquia Os comportamentos estão organizados numa hierarquia fixa de subsunção.

Prioridade As ações são selecionadas de acordo com uma prioridade associada que varia ao longo da execução

Fusão As respostas são combinadas numa única resposta por composição

Comportamentos sem memória Acoplamento Perceção-Ação

- Depende fortemente das capacidades sensoriais
- Depende das caraterísticas do ambiente

Um exemplo deste tipo de comportamento é os comportamentos de exploração

Problemas

- Exploração - Necessidade de evitar o passado
- Ótimos locais - Por exemplo, os veículos de Braitenberg ficam presos nos cantos, incapazes de dar a volta
- Comportamentos cíclicos - Movimentação cíclica perante determinadas configurações de alvos e obstáculos

Para limitar estes problemas é necessário uma manutenção de estado.

Arquitetura Reativa com Memória Numa arquitetura deste tipo, as reações dependem não só das perceções, mas também da memória das perceções anteriores para gerar as ações.
Para este efeito é necessário manter uma memória interna, a qual é atualizada a partir das perceções e das reações ativadas, influenciando essas mesmas reações, bem como as ações geradas.

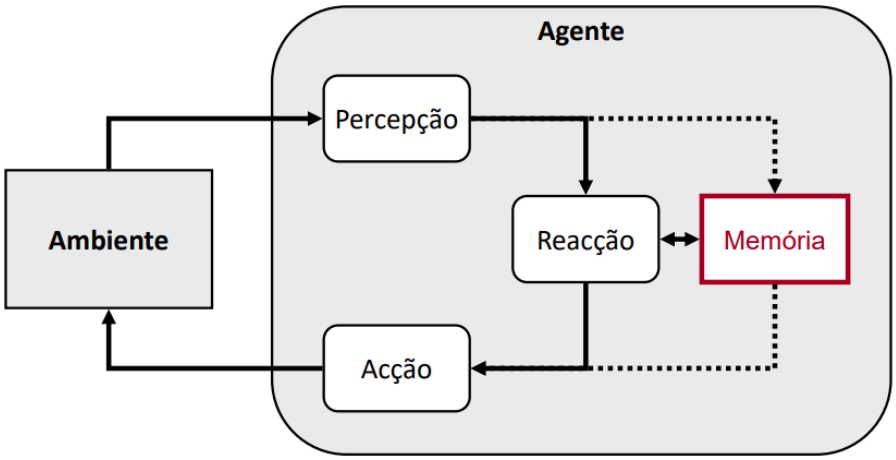


Figura 9: Arquitetura de um Agente Reativo com Memória

2.5 Raciocínio Automático e Tomada de Decisão

A resolução automática de problemas é uma área da inteligência artificial orientada para a resolução de problemas por meios algorítmicos, através do **raciocínio automático**.

Raciocínio Automático Refere-se à capacidade de um sistema computacional resolver de forma automática um problema com base numa representação de conhecimento do respetivo domínio, produzindo uma solução a partir de diversas alternativas possíveis.
Este tipo de raciocínio envolve dois tipos de atividades principais:

Exploração de opções

- Raciocínio prospetivo (Antecipação do que pode acontecer)
- Simulação interna do domínio do problema (representação interna do domínio do problema)

Avaliação de opções

- Custo (recursos necessários)
- Valor (Ganho ou perda)

Noções envolvidas neste tipo de raciocínio:

Estrutura Os estados abstraem os aspetos estruturais de um problema ou sistema.

Estados Representam configurações.

Dinâmica Os operadores abstraem as transformações (dinâmica) que podem ocorrer no estado de um problema ou sistema.

Operadores Representam transformações.

Espaço de Estados Conjunto de estados e de transições de estado.

Problema Junção do estado inicial, os operadores disponíveis e objetivo final.

Mecanismo de Raciocínio Exploração de opções possíveis para encontrar uma solução através de simulação prospetiva.

Métodos de Procura

- Procura em profundidade (Depth-First Search)
- Procura em Largura (Breadth-First Search)
- Procura em Grafo
- Procura Melhor-Primeiro (Best-First)

O tipo de Procura Melhor-Primeiro têm 3 diferentes variantes:

Procura de Custo Uniforme Minimização de custo acumulado até cada nó explorado.

Procura Sôfrega Minimização da estimativa de custo para atingir o objetivo, não tem em conta o custo do percurso explorado e é uma solução sub-ótima.

Procura A* (heurística admissível) Minimização de custo global.

h(n) Estimativa de custo(heurística) para a partir do estado inicial atingir o estado objetivo e é utilizada pelas procuras Sôfrega e A*.

2.6 Arquitetura de Agentes deliberativos

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

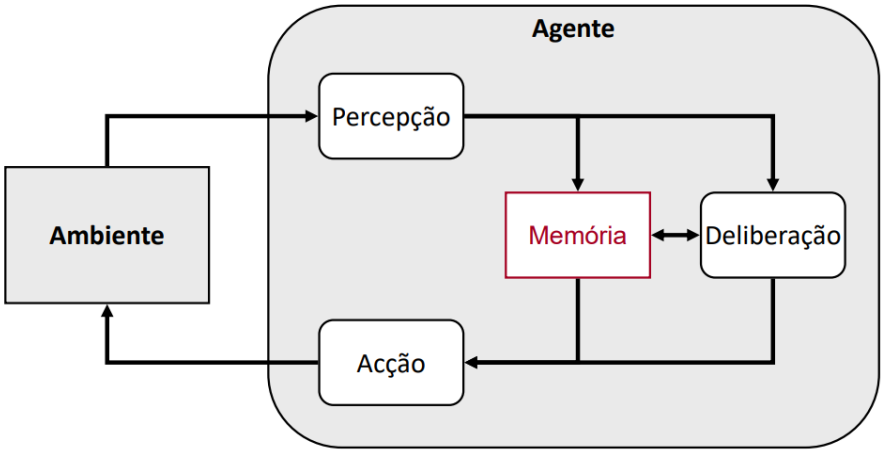


Figura 10: Arquitetura de um agente deliberativo

Raciocínio Prático No contexto do agente autónomo que opera num determinado ambiente, é particularmente relevante o raciocínio orientado para a ação, designado de Raciocínio prático. Este tipo de raciocínio é orientado para a ação.

Elementos de suporte

- Representação dos objetivos a atingir
- Representação das ações realizáveis
- Representação do mundo (ambiente)

E o resultado deste raciocínio é os planos de execução.

O processo geral de tomada de decisão e ação é o seguinte

1. Observar o mundo, gerando perceções
2. Atualizar o modelo do mundo, com base nas perceções
3. Deliberar o que fazer, gerando um conjunto de objetivos
4. Planear como fazer, gerando um plano de ação
5. Executar o plano de ação

Este tipo de raciocínio tem também os seus problemas caso haja recursos computacionais limitados como falta de memória e tempo de computação demorado, caso haja dinamismo do ambiente o resultado do raciocínio pode não ser consistente com a situação do ambiente podendo ser necessário a reconsideração de opções.

Tipos de Processos deliberativos

- Planeamento automático

Objetivo Gerar sequências de ação, designadas por planos, para concretização de objetivos pré-definidos.

O planeamento é realizado com base em métodos de raciocínio automático, como procura em espaços de estados, ou processos de decisão de Markov.

Modelo de Planeamento:

- Estado inicial do problema
- Conjunto de estados válidos
- Conjunto de operadores definidos
- Objetivos a atingir

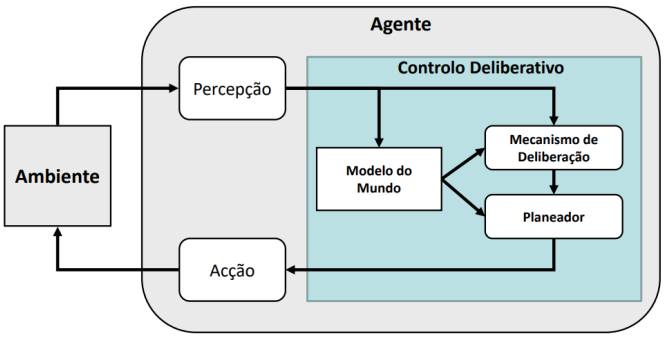


Figura 11: Mecanismo de planeamento de um agente

2. Planeamento com base em PDM

Este tipo de planeamento é baseado numa representação do problema sob a forma de um processo de decisão de Markov. Produz como resultado uma política de ação, a qual define para cada estado possível a respetiva ação a realizar.

Utilidade e Política O raciocínio automático com base em processos de decisão de Markov envolve o cálculo da função de utilidade (valor), que define o valor associado a cada estado do problema, a qual serve de suporte ao cálculo da política de ação, nomeadamente, a política ótima, que determina a seleção de ações que maximizam o valor de ação em cada estado.

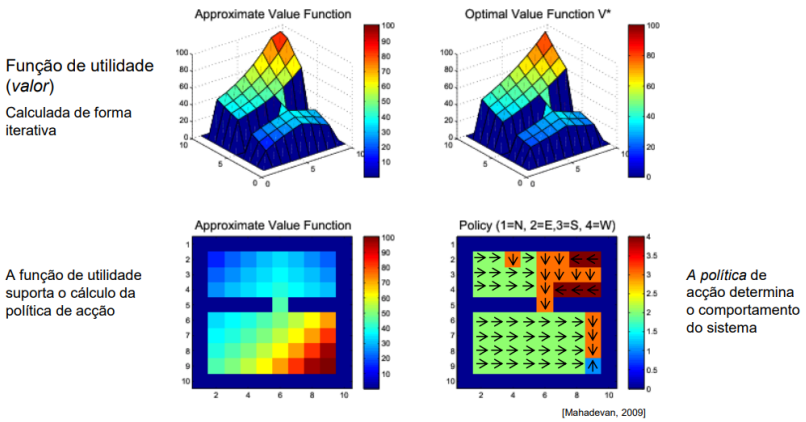


Figura 12: Funções de utilidade e Política

2.7 Aprendizagem por Reforço

Aprendizagem = Melhoria de desempenho, para uma dada tarefa, com a experiência.

A aprendizagem por reforço é a aprendizagem a partir da interação com o ambiente

- Estado
- Ação
- Reforço (Ganho/Perda)

Ou a aprendizagem de comportamentos

- O que fazer
- Relação entre situações e ações

O agente vai aprendendo ao longo que interage com o ambiente. Existe, no entanto, um dilema que é preciso tomar atenção, o **Dilema Explorar/Aproveitar**

Exploração Escolher uma ação que permita explorar o mundo para melhorar a aprendizagem.

Aproveitamento

- Escolher a ação que leva à melhor recompensa de acordo com a aprendizagem
- Ação Sôfrega (Greedy)

Este tipo de aprendizagem é muito utilizado em diversas áreas como a robótica, jogos (treinar os seus NPCs para tomarem melhores decisões), finanças, e condução autónoma.

3 Projeto Realizado

Este projeto foi dividido em duas partes, a primeira utilizando a linguagem "Java"e com o propósito de entender os UMLs e como os ler e utilizar e o objetivo de criar um jogo com ambiente onde a personagem tem por objetivo registar a presença de animais através de fotografias. A segunda parte deste projeto, realizada na linguagem "Python"e esta com o propósito de estudar, aprender e utilizar a inteligência artificial para diversos propósitos.

3.1 Primeira Parte

Nesta primeira parte pretende-se implementar um jogo com uma personagem virtual que interage com um jogador humano.
Este jogo consiste num ambiente onde a personagem tem como principal objetivo registar a presença de animais através de fotografias.
É iniciado com a personagem numa estado de procura de animais. Ao receber um estímulo para um determinado estão dentro do seu conjunto vai realizar uma ação para um determinado evento. A sua interação é realizada em modo de texto.

Foi inicialmente realizado código para dar inicio ao jogo, execução do jogo e estudado a arquitetura deste jogo. Tudo isto com base em **Sequence Diagrams** e **Class Diagrams** de modo a saber os métodos de cada classe e como interagiam umas com as outras.

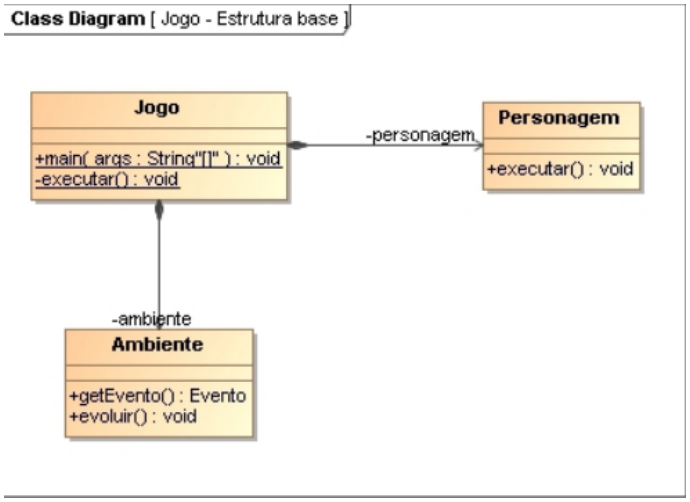


Figura 13: Estrutura base do projeto

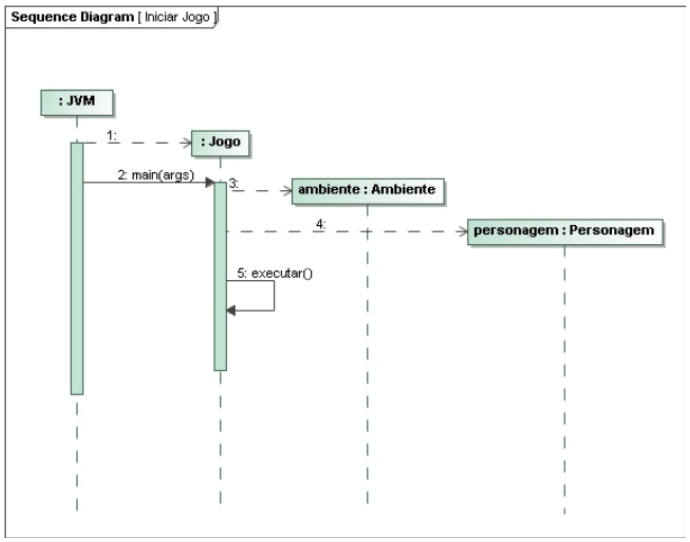


Figura 14: Sequence Diagram do Iniciar Jogo

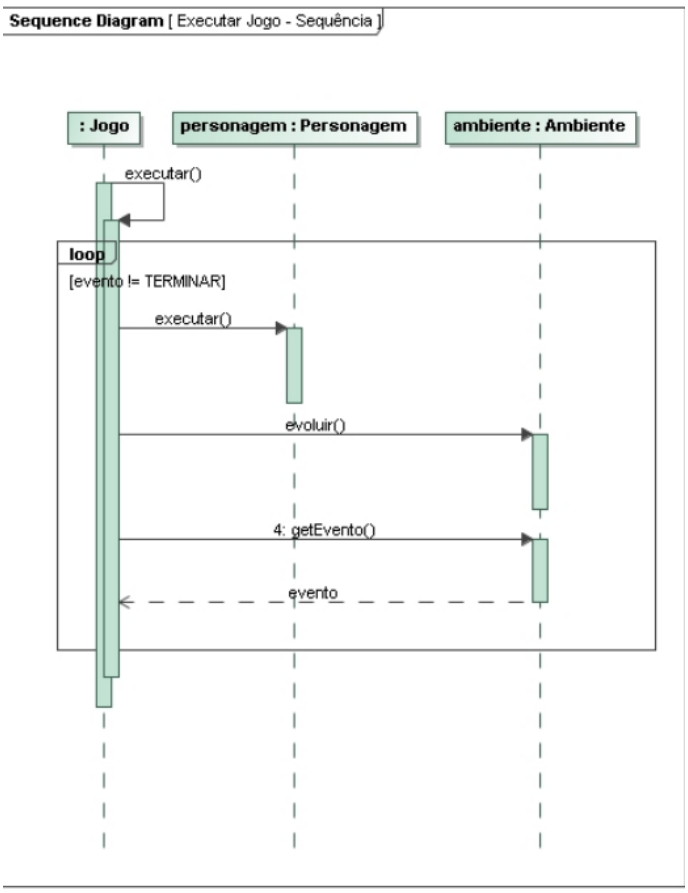


Figura 15: Sequence Diagram do Executar jogo

Com a evolução do desenvolvimento deste projeto foi adicionado uma enumeração com os eventos possíveis de acontecerem, uma enumeração que vai ser utilizada num novo método da classe "Ambiente"chamado *evoluir()* e assim evoluindo a estrutura e a complexidade deste projeto.

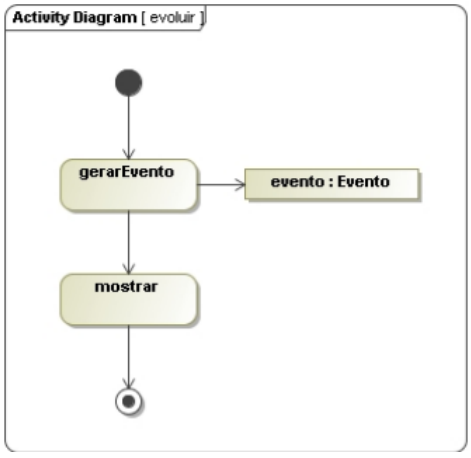


Figura 16: Activity Diagram do método Evoluir

Na classe "Personagem"é onde se desenvolve o modelo conceptual de uma arquitetura de um agente inteligente tendo esta classe os 3 métodos necessários para esse propósito e criando uma nova classe "Controlo"que vai ter o método *processar()*

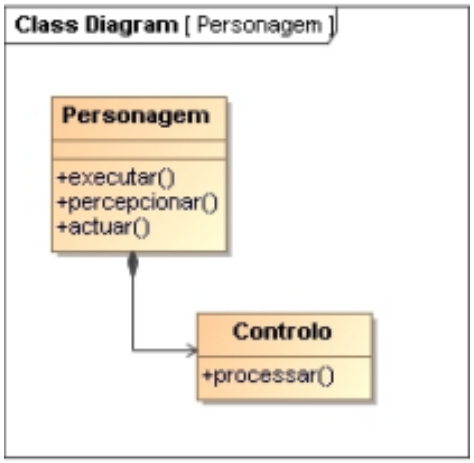


Figura 17: Class Diagram da classe Personagem

Foi então desenvolvida uma máquina de estados que passaria a ser utilizada pelo jogo de modo a poder criar a teoria por detrás deste jogo e como cada estado interagia com o outro tendo em conta um evento dando uma ação à personagem.

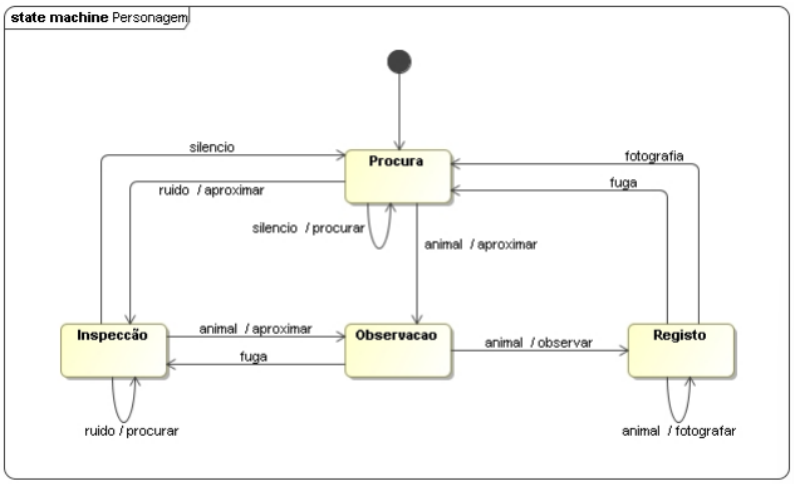


Figura 18: State Machine da Personagem

Para esta máquina de estados ser funcional no projeto foi necessário criar 3 classes: "Maquina-Estados", "Estado"e "Transição"ficando o Class Diagram dessas classes desta forma:

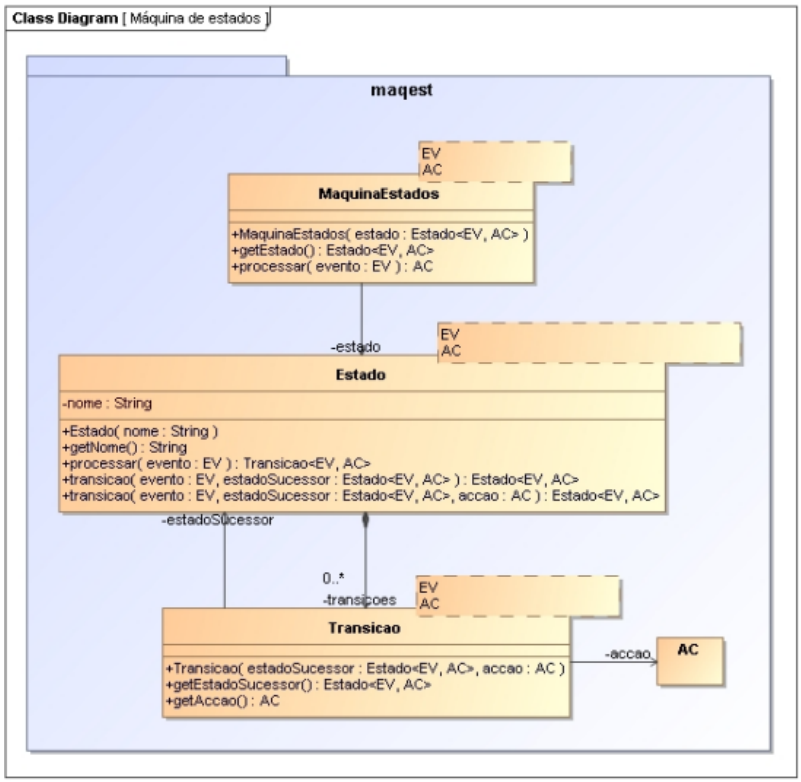


Figura 19: Class Diagram package maqest

Ao desenvolver todas estas classes teremos o projeto a funcionar como se pretende.

```
Maquina de Estados --> Procura
Escolha o evento
r
Evento --> RUIDO
Maquina de Estados --> Inspeccao
Estado da ação --> APROXIMAR
Escolha o evento
a
Evento --> ANIMAL
Maquina de Estados --> Observacao
Estado da ação --> APROXIMAR
Escolha o evento
o
Evento --> FOTOGRAFIA
Maquina de Estados --> Observacao
Escolha o evento
f
Evento --> FUGA
Maquina de Estados --> Inspeccao
Escolha o evento
t
Evento --> TERMINAR
```

Figura 20: Exemplo do funcionamento do jogo

Pode-se observar nesta imagem um exemplo da simulação deste jogo onde o utilizador introduz um evento e o jogo, de acordo ao produzido na maquina de estado, atribui a ação que a personagem tem de realizar.

3.2 Segunda Parte

O objetivo desta segunda parte é a criação e a utilização de agentes inteligentes de acordo com as diversas arquiteturas lecionadas nesta cadeira.
Foi fornecido pelo docente uma biblioteca chamada de Simulador de Ambiente de Execução(SAE). Esta biblioteca tem o propósito de poder simular um ambiente virtual em que o agente possa interagir e que permita ao utilizador visualizar o processamento do agente inteligente.

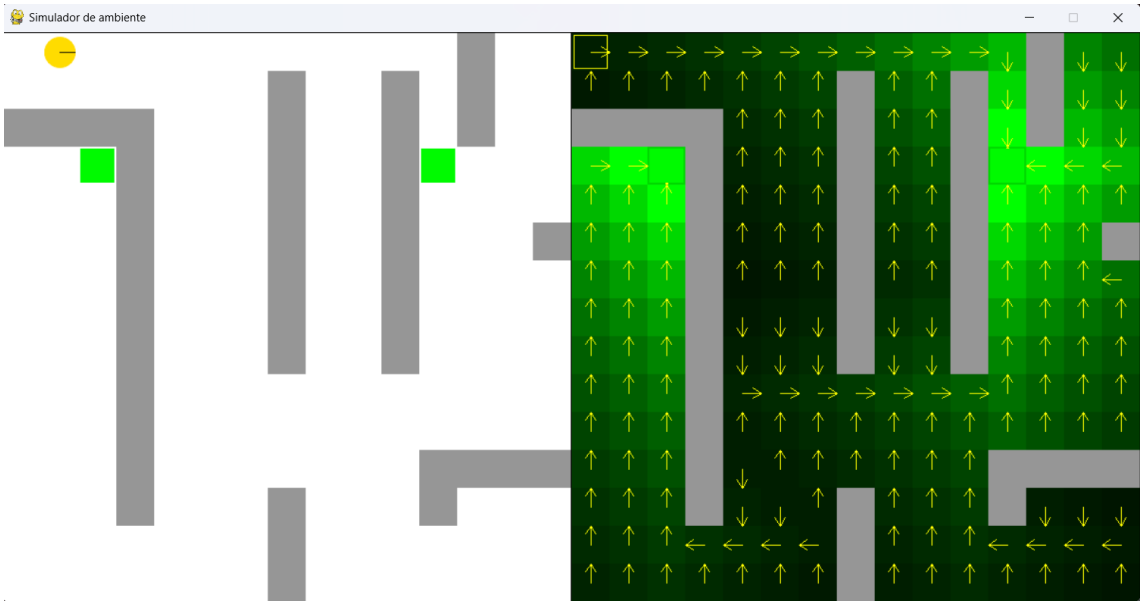


Figura 21: Exemplo do uso da biblioteca SAE

Nas primeiras três aulas destinadas a este projeto foi desenvolvido código de acordo com uma arquitetura de agentes reativos, criando classes para a reação, estímulo, resposta do agente, esta utilizando a classe Accao da biblioteca SAE, uma classe abstrata para o comportamento e uma classe para o comportamento composto que recebe como herança a classe Comportamento.

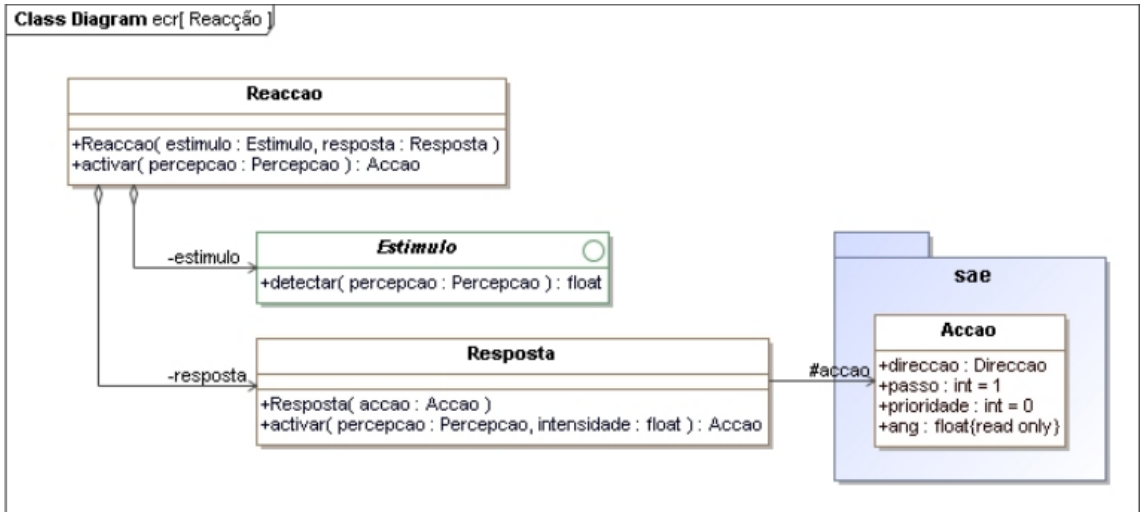


Figura 22: Class Diagram ecr

Classe abstrata Estimulo Esta é uma classe abstrata com o método detetar. Este método ao ser realizado deverá detetar um estimulo com a perceção dada como atributo e retornar um valor de intensidade.

Classe Resposta Nesta classe é definido a resposta aos estímulos ao utilizar o método ativar para atribuir uma ação ao agente.

Classe Reaccao Esta classe recebe no seu construtor um estimulo e uma resposta e tem um método chamado activar() com o propósito de ativar uma ação caso exista um estimulo.

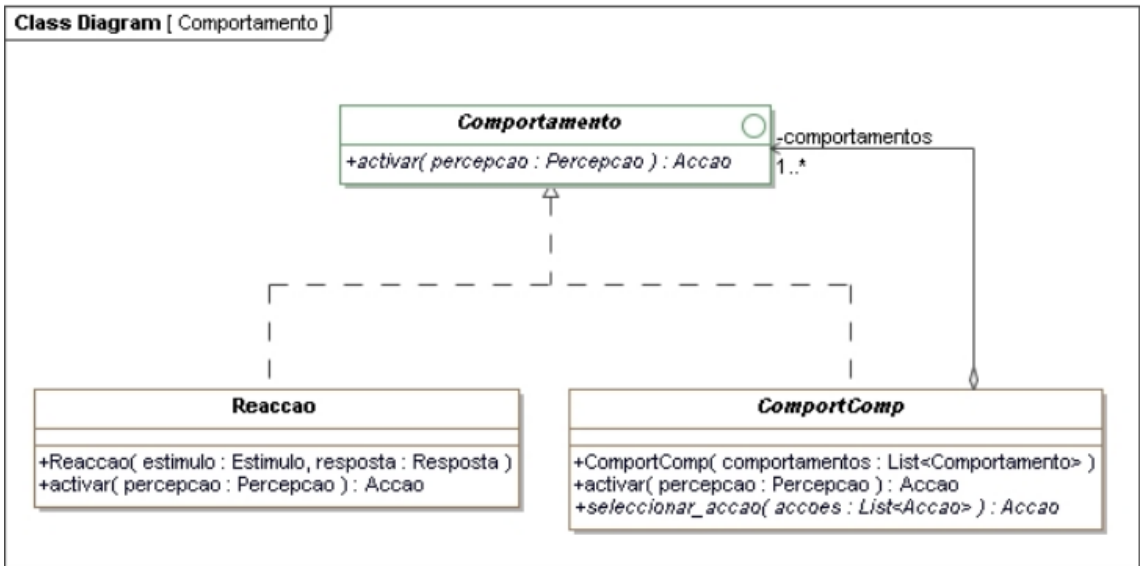


Figura 23: Class Diagram Comportamento

Classe abstrata Comportamento Esta classe vai ter uma classe abstrata chamada activar que ao ser desenvolvida deverá retornar uma ação.

Classe ComportComp Esta classe tem um contrato funcional com a classe Comportamento e recebe um array de comportamentos como atributo. O método activar verifica todos os comportamentos desse array e caso esse comportamento exista adiciona a um novo array. No final retorna esse novo array com os comportamentos ativados.

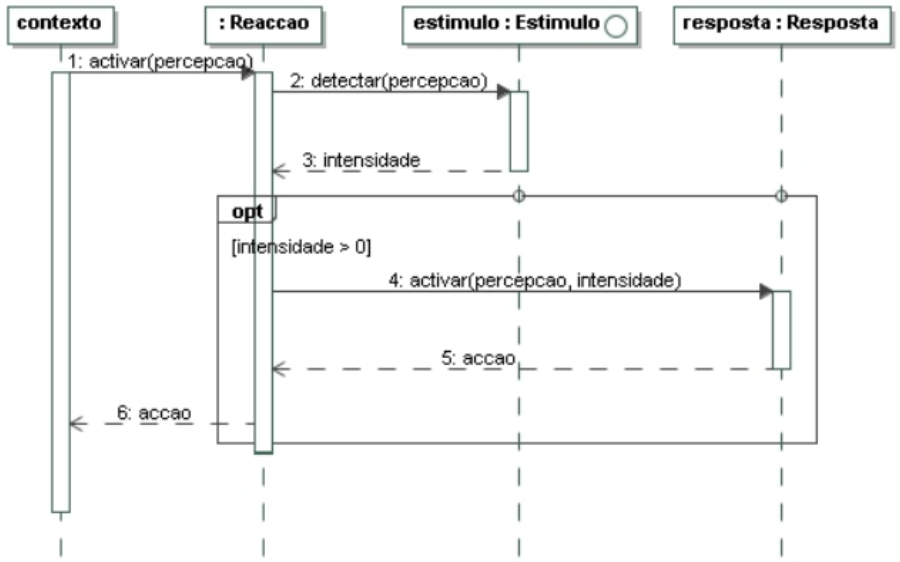


Figura 24: Modelo de interação destas classes

Tipos de Comportamento Nos agentes reativos há dois possíveis tipos de comportamentos, **Hierarquia** e **Prioridade**. Para estes dois comportamentos foram criadas classes com um único método, o *seleccionar accao()* que com uma lista de ações escolhe a ação principal para o seu tipo de comportamento.
Foi também desenvolvido uma classe, "RespostaMover", uma classe herança de "Resposta"que cria uma instancia de uma ação com a direção passada como atributo.

Com o código desenvolvido até agora e com a criação da classe "ControloReact"que é utilizada para processar uma perceção foram criadas 3 diferentes reações:

- Aproximar
- Evitar
- Explorar

Reação Explorar Esta reação utiliza apenas uma classe "Explorar". Esta classe utiliza apenas um método, *activar(percepcao)* com o objetivo de gerar uma ação que representa uma movimentação aleatória de uma das 4 direções possíveis do agente no ambiente virtual da biblioteca SAE.

Reação Aproximar Esta reação dá uso a 3 classes, "AproximarAlvo"com a função de criar internamente as 4 instâncias da classe "AproximarDir"com as direções necessárias e utilizar o construtor da superclasse "Prioridade", "AproximarDir"que é uma reação com o estímulo alvo e aciona uma "RespostaMover"e "EstimuloAlvo"que deteta as distancias aos alvos e quanto menor essa distância maior a prioridade.

Reação Evitar Esta reação tem o objetivo de evitar obstáculos de um ambiente, como por exemplo barreiras e dá uso a 4 classes, "EvitarDir"com a função de utilizar o construtor da sua superClasse "Reaccao"para instanciar um estimulo da classe "EstimuloObst"e uma resposta para criar uma reação do tipo Evitar, "EstimuloObst"com a função de detetar se algum dos elementos à volta do agente é ou não um obstáculo e retornar a intensidade desse estímulo casa exista, "RespostaEvitar"tendo o objetivo oposto da classe "RespostaMover"onde ativa uma ação no caso de estar em contacto com um obstáculo e detetar que existem direções livres para o quais se pode mover, neste método são usados dois métodos necessários para o seu correto funcionamento, *direccaoLivre()* que deteta as direções livres para as quais o agente se pode mover e o método privado *alterarDireccao()* que altera a direção para o qual o agente se tem de movimentar.

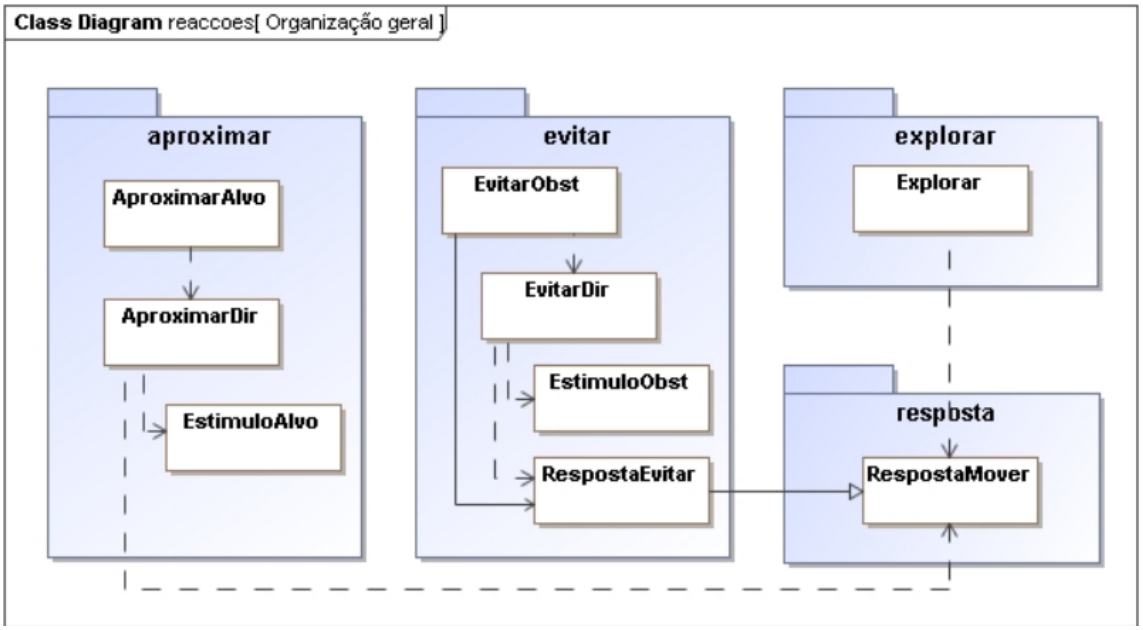
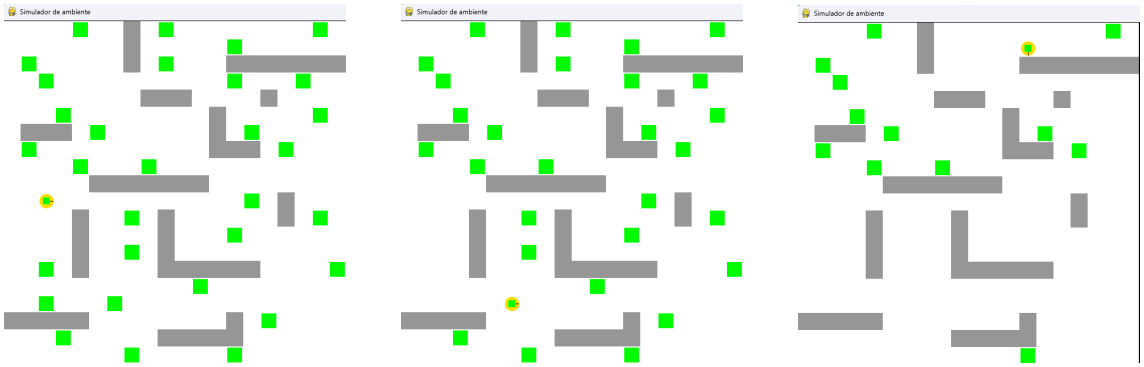


Figura 25: Class Diagram Organização geral das reações

Classe Recolher Esta classe têm o propósito de reunir as três reações mencionadas anteriormente para poderem ser utilizadas no simulador seguindo a base da hierarquia.



(a) Agente no seu estado Inicial (b) Uns passos após o inicio (c) Após vários passos

Figura 26: Simulação de um agente com as três reações desenvolvidas

Raciocínio Automático com Base em PEE

Destinada à aprendizagem do raciocínio automático foi necessário o desenvolvimento de uma package chamada "mod"onde se irá incluir classes para o estado, operador e problema.

Estado

- Identificação única por valor(em função da informação de estado)

Operador

- Aplicado a um estado, gera um novo estado
- Define custo de transição de estado

Problema

- Estado inicial
- Operadores
- Função de teste de objetivo

Com estas classes desenvolvidas foi então estudado o mecanismo de procura em espaços de estados.

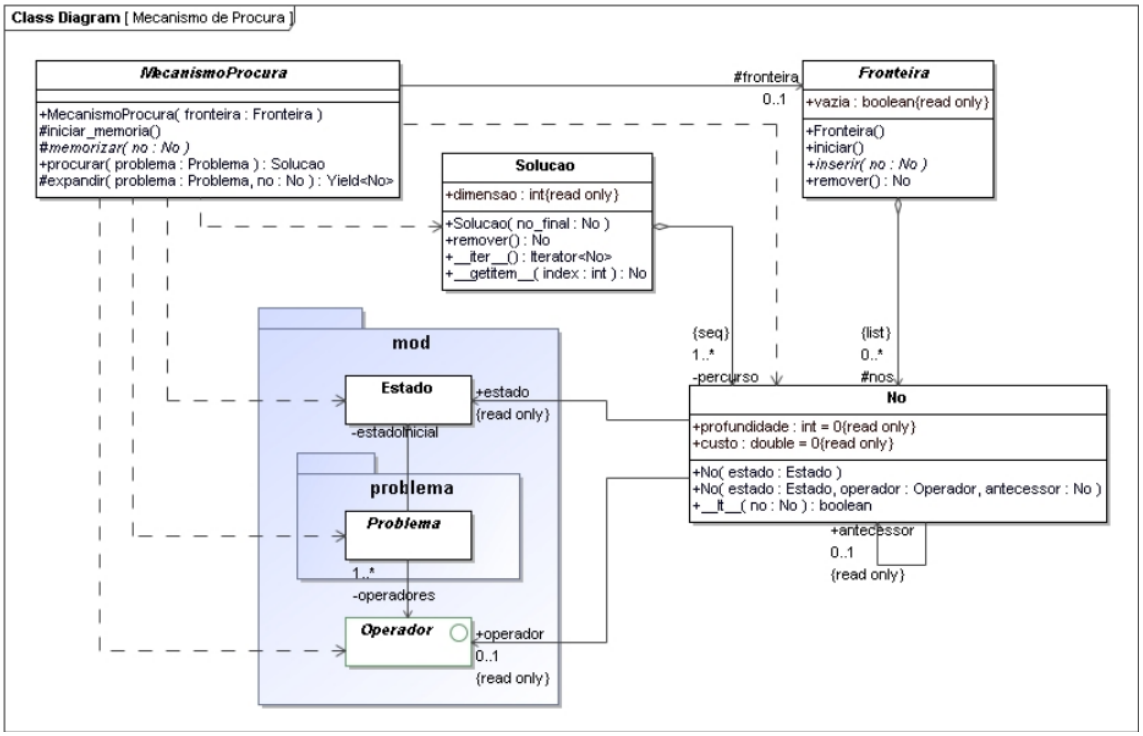


Figura 27: Class Diagram do Mecanismo de Procura em espaços de estados

Ao verificar a imagem de cima pode-se observar que é necessária a criação de 4 classes para o funcionamento correto do mecanismo de procura

Classe MecanismoProcura Esta classe vai ter o objetivo de procurar uma solução para determinado problema a utilizar uma fronteira de exploração para memorizar e gerir os nós explorados. Para conseguir ser realizado foi desenvolvida a função procurar com esse objetivo tendo os seguintes passos.

1. Iniciar a memória do mecanismo
2. Criar um nó inicial com o estado inicial do problema dado
3. Inserir o nó na fronteira
4. enquanto a fronteira não for vazia remover o nó da fronteira
5. se o estado do nó for o objetivo retornar a solução com esse nó
6. se não for então utilizar o método **expandir()** desta classe num For e memorizar os nós sucessores da fronteira

Classe No Esta classe é utilizada para definir os nós da árvore de procura, dando-lhes um estado, operador, nó antecessor caso exista, a profundidade do nó na árvore de procura e o custo do percurso até chegar a esse nó.

Classe Solucao Esta classe tem o propósito de representar um percurso de nós correspondentes à solução do problema sendo possível verificar a sua dimensão, remover nós dessa lista, e verificar os nós totais desse percurso.

Classe Fronteira Esta classe permite inserir e remover nós de forma ordenada, iniciar a fronteira de nós e verificar o número de nós ainda existentes.

Com estas classes desenvolvidas é possível criar outras mais direcionadas a cada tipo de procura individual, como as classes "ProcuraProfundidade", "ProcuraGrafo", "ProcuraLargura", "ProcuraMelhorPrim", "ProcuraCustoUnif", "ProcuraInformada", "ProcuraAA"e "PorucraSofrega", classes que recebem herança da super classe "MecanismoProcura".

Com estas classes desenvolvidas e as classes de suporte a estas procuras como os avaliadores, as fronteiras LIFO, FIFO e Prioridade foi então realizado uma aplicação de teste para um planeador de trajetos entre localidades.

Conceitos principais envolvidos

Planeador de Trajetos

Permite planear um trajeto, gerando uma solução, com base na seguinte informação:

- Ligações, lista de ligações entre localidades
- Localidade inicial
- Localidade final

Permite mostrar o trajeto correspondente a uma solução

Ligação Define a origem, destino e custo de uma ligação

Problema de planeamento de trajetos

Define um problema de planeamento de trajetos, com base na seguinte informação

- Ligações, lista de ligações entre localidades
- Localidade inicial
- Localidade final

Estado de representação de uma **localidade**

Operador de representação de realização de uma ligação

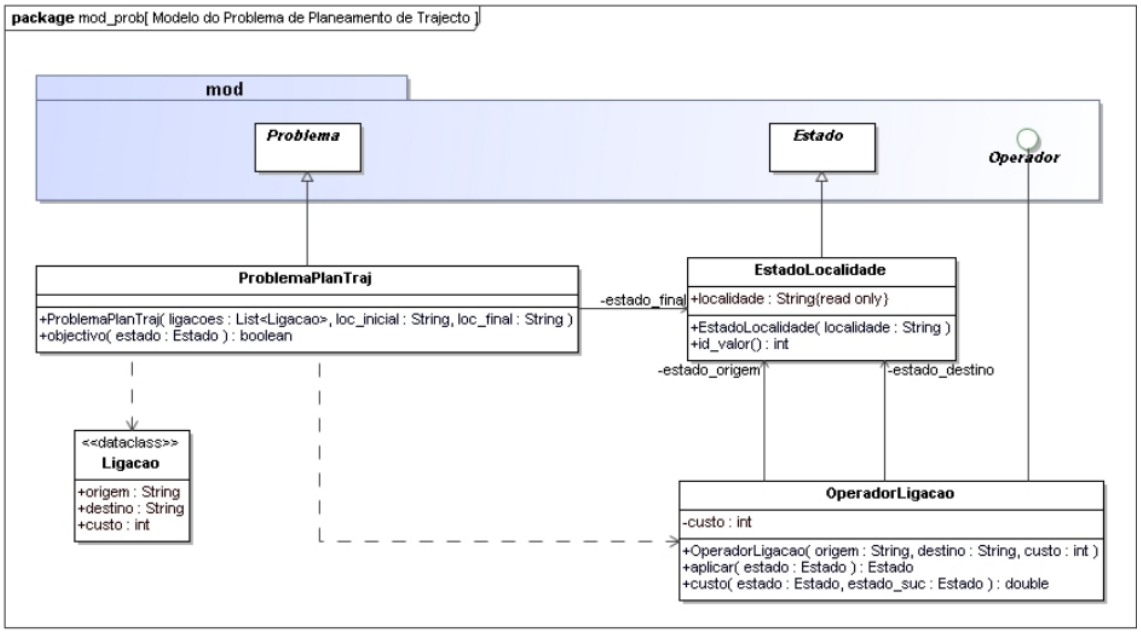


Figura 28: Class Diagram da aplicação desenvolvida

Foi então necessário a criação destas classes para o correto funcionamento da aplicação.

ProblemaPlanTraj Herança da classe "Problema", esta classe recebe as ligações existentes no problema, a sua localidade inicial e a final e o método **objetivo()** que verifica se o estado atual em que o agente se situa é a localidade final ou não.

Ligacao Tipo dataclass dá tipos aos atributos utilizados na aplicação, string para a origem e destino e int para o custo de cada movimento.

EstadoLocalidade Herança da classe "Estado", esta classe terá o objetivo de criar um estado para cada localidade existente dando-lhe um id único.

OperadorLigacao Herança da classe "Operador", esta classe tem o objetivo de criar um operador entre duas localidades com um certo custo e aplicar essa movimento entre localidades caso o estado atual seja igual ao estado de origem de dado operador.

PlaneadorTrajeto Esta classe utiliza um dos tipos de procura para poder ser realizado o objetivo da aplicação.

```
Resultados Obtidos
ProcuraCustoUnif()
Loc-0 : None
Loc-1 : Loc-0 -> Loc-1
Loc-3 : Loc-1 -> Loc-3
Loc-5 : Loc-3 -> Loc-5
Loc-4 : Loc-5 -> Loc-4

ProcuraProfLim()
Loc-0 : None
Loc-2 : Loc-0 -> Loc-2
Loc-4 : Loc-2 -> Loc-4

ProcuraProfIter()
Loc-0 : None
Loc-2 : Loc-0 -> Loc-2
Loc-4 : Loc-2 -> Loc-4

ProcuraLargura()
Loc-0 : None
Loc-2 : Loc-0 -> Loc-2
Loc-4 : Loc-2 -> Loc-4
'''
```

Figura 29: Resultados obtidos com as diferentes pesquisas

Como se pode observar pela imagem acima apenas a procura por custo uniforme obteve resultados diferentes aos outros tipos de procuras pois o critério de exploração é o custo de cada operador enquanto que nas outras o critério de exploração é a menor profundidade possível de nós.

Controlo Deliberativo

Nesta ultima parte do projeto foi desenvolvido um arquitetura de agentes deliberativos sendo criada uma nova package destinada às classes do controlo deliberativo.

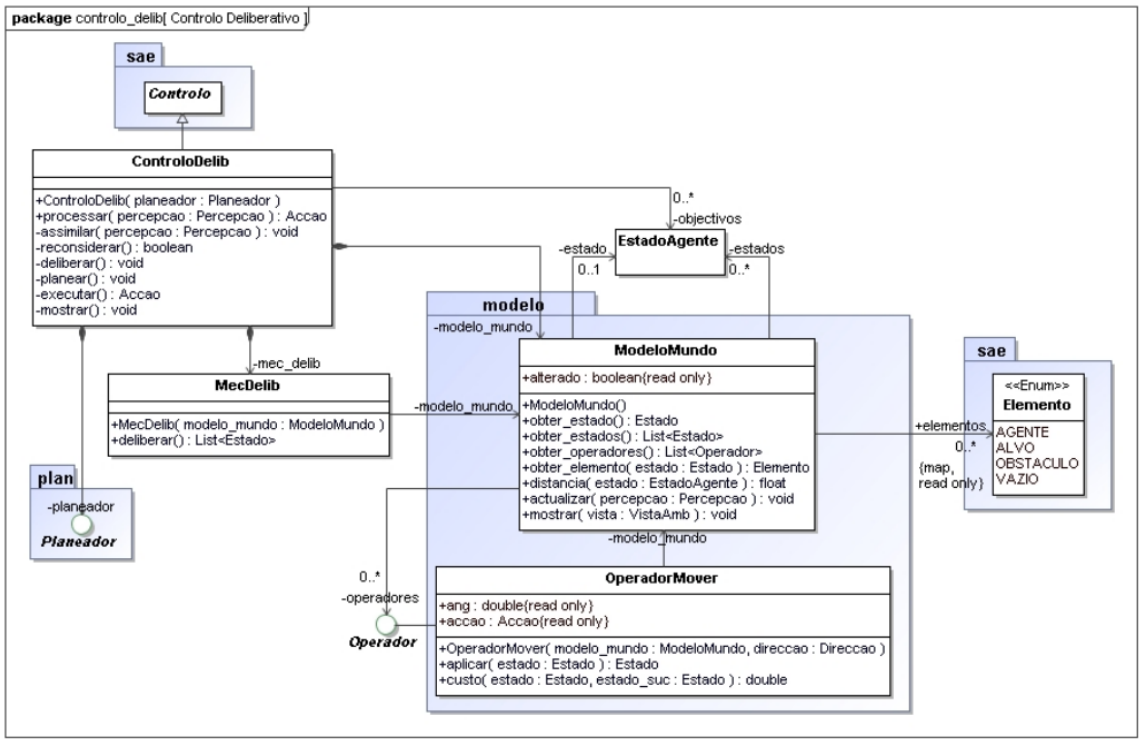


Figura 30: package controlo deliberativo

MecDelib Esta classe tem o objetivo de criar o mecanismo da arquitetura deliberativa tendo o método **deliberar()** onde se vai gerir um conjunto de objetivos, estados objetivo do agente, e ordenar essa lista de objetivos por distância até ao agente.

ControloDelib Esta classe vai ter os métodos necessários para o processamento de uma perceção.

- assimilar() - atualizar o modelo mundo com a nova informação da perceção
- reconsiderar() - reconsiderar se existiu alterações no modelo mundo
- deliberar() - criação dos novos objetivos do agente a usar a função **deliberar()** da classe "MecDelib"
- planear() - Gerar um novo plano para a concretização desses planos
- executar() - Utilizar esse novo plano para ir buscar a nova ação do agente

ModeloMundo Esta classe irá ter todas as instâncias necessárias para o correto funcionamento da arquitetura. Têm métodos necessários para ir buscar o estado atual do agente, os estados existentes, os possíveis operadores do agente, o elemento existente no estado atual do agente e métodos para saber a distancia atual entre o estado do agente e a posição de um outro estado e o método **actualizar()** que com uma perceção terá os próximos passos:

1. atualizar o atributo estado no construtor com um novo agente e essa posição
2. se os elementos da perceção forem diferentes aos elementos pré-existentes - atualizar a estrutura interna do agente com os novos elementos para o agente e o conjunto de estados com um novo conjunto

OperadorMover Herança da super classe "Operador", esta classe tem o objetivo de atribuir operadores aos estados com uma distancia do estado atual até a outro estado usando o método **translacao()** que cria uma nova posição, com a distancia e ângulo desde a posição do estado inicial.

EstadoAgente Esta classe, herança da super classe "Estado", tem um atributo posição associado e um id valor único a esse estado.

Após o desenvolvimento destas classes foram criadas três classes abstratas, "ModeloPlan", "Plano"e "Planeador", estas três classes vão ser usadas para dois tipos de planeamento diferentes, *Planeador com base em PEE* e *Planeamento com base em PDM*

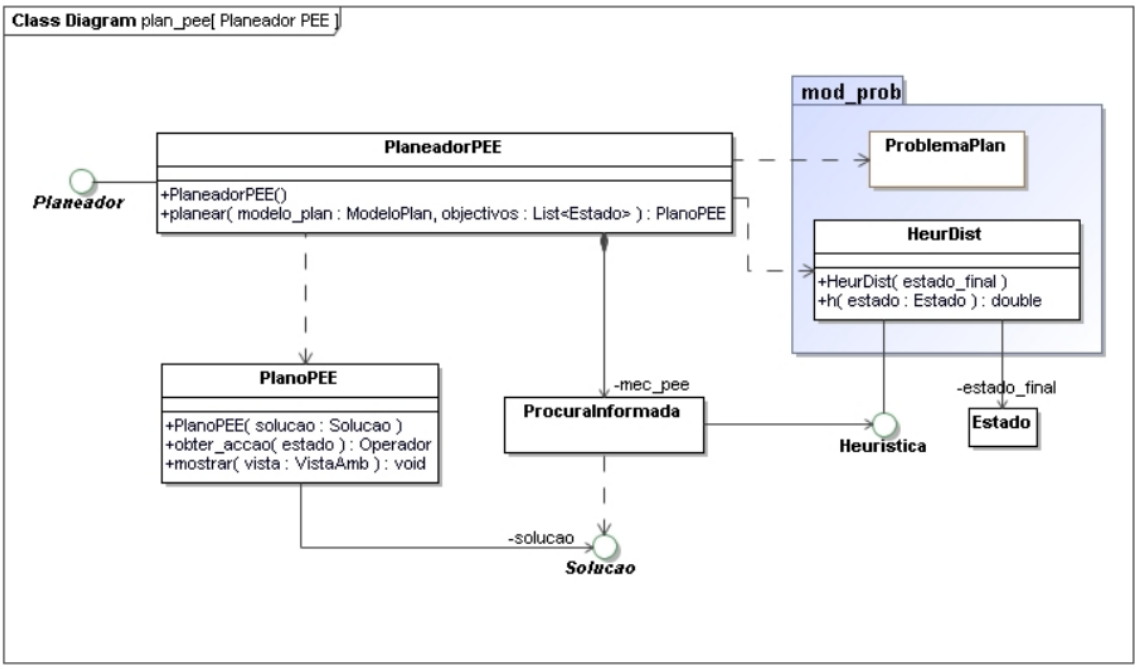


Figura 31: Class Diagram Planeador PEE

PlanoPEE Esta classe é utilizada para pesquisa por espaço de estados, utiliza o método obter-accao para obter o operador do estado caso este esteja na solução:

1. Verificar a dimensão da solução e enquanto for maior que 1:
2. confirmar que o estado do primeiro nó da solução é igual ao estado dado pelo método
3. se for igual remove-se o primeiro nó com o operador nulo
4. no final retornar o operador do primeiro nó da solução atualizada

PlaneadorPEE Esta classe utiliza 3 diferentes tipos de procura:

- ProcuraAA
- ProcuraSofrega
- ProcuraCustoUnif

e planeia uma solução para o agente realizar dado os seus objetivos, o modelo-plan e o tipo de procura utilizado.

Antes de poder ser desenvolvido um planeador para o processo de decisão de Markov foi necessário criar classes que realizam esse processo de decisão, "PDM", "ModeloPDM"e "MecUtil"

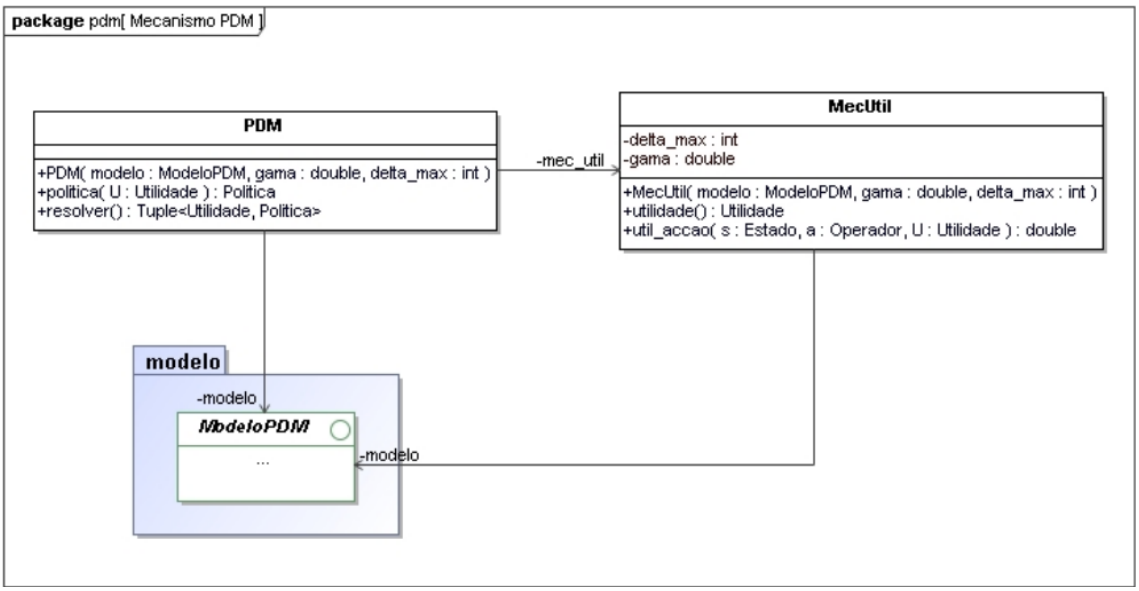


Figura 32: Class Diagram Mecanismo PDM

MecUtil Nesta classe existem vários métodos para calcular a utilidade de um conjunto de estados de um problema e para verificar a utilidade de uma ação, este método utiliza a equação do cálculo da utilidade de estado.

function $U_{acção}(s,\ a,\ U)$:

return $\sum_{s'} T(s,a,s')[R(s,a,s') + \gamma U[s']]$

Figura 33: Equação do cálculo da utilidade

PDM Esta classe é onde se calcula a política do modelo **politica()** dando uma utilidade de um estado e um método **resolver()** onde se calcula uma utilidade e uma política retornando um tuplo.

ModeloPDM Esta classe abstrata dá as informações necessárias para se poder desenvolver um modelo PDM para um certo propósito tendo os métodos **S()** para guardar o conjunto de estados, **A** para guardar o conjunto de ações possíveis de um certo estado S, **T()** para guardar a probabilidade de transição de um estado para outro através de uma ação, **R()** para guardar o retorno esperado na transição de s para 2' através de a, **Suc()** para ir buscar os estados sucessores de um conjunto de estados.

Com estas classes definidas pode-se então desenvolver um planeadorPDM

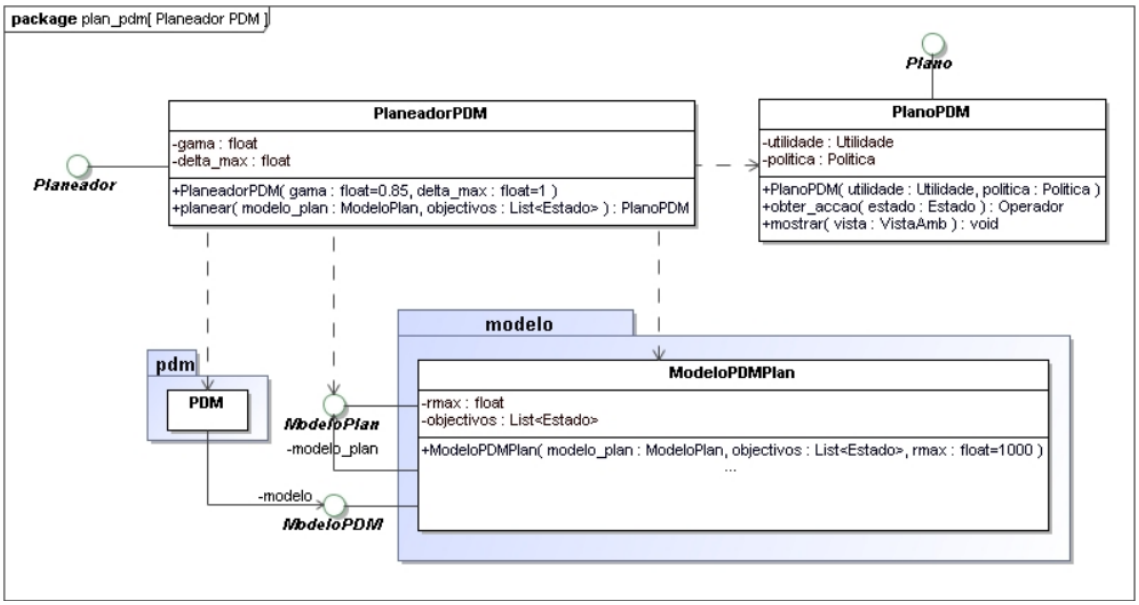


Figura 34: Class Diagram Planeador PDM

PlaneadorPDM Esta classe irá ser utilizada para criar um planoPDM com uma utilidade e politica da classe PDM.

PlanoPDM Esta classe é utilizada para saber a ação que o agente tem de realizar dada uma politica.

ModeloPDMPlan Esta classe é utilizada para descobrir os estados do agente, os seus objetivos e as recompensas para cada ação.

Com estas classes desenvolvidas foi possível então realizar aplicações para cada um dos planeadores iniciando com o teste ao planeador com base em PEE

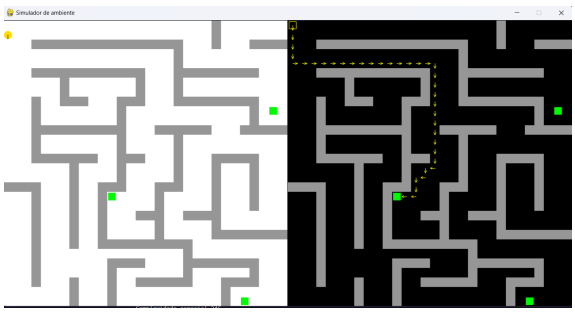


Figura 35: Estado inicial do agente

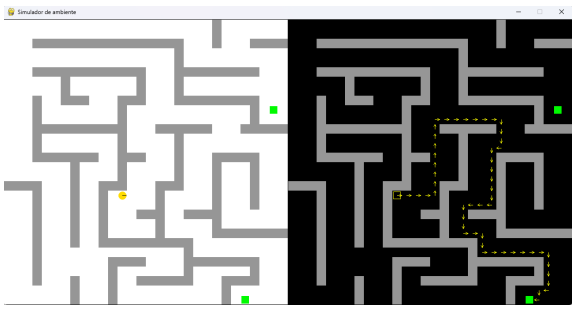


Figura 36: Após ter chegado ao primeiro objetivo

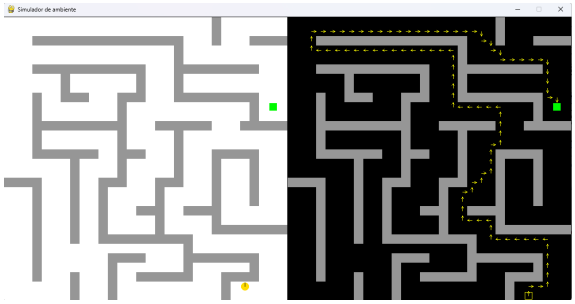


Figura 37: Após ter chegado ao segundo objetivo

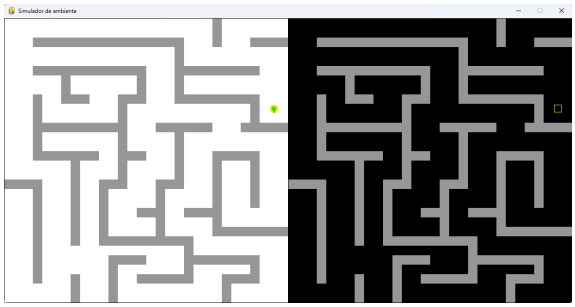


Figura 38: Após ter chegado ao objetivo final

Pode-se observar por estas imagens o agente em diferentes posições e o caminho que teve de tomar até chegar aos objetivos pré-definidos. Neste exemplo foi utilizado o tipo de procura AA sendo que com outros tipos podem existir diferentes caminhos até aos objetivos.

E agora com o teste ao planeador baseado em PDM

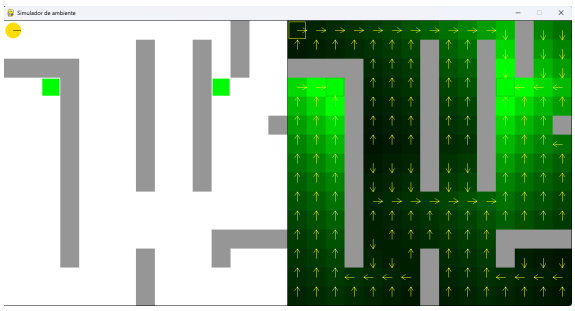


Figura 39: Estado inicial do agente

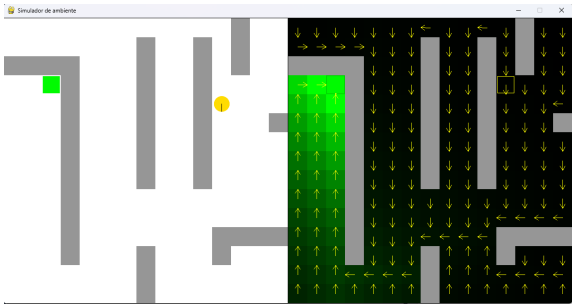


Figura 40: Após ter chegado ao primeiro objetivo

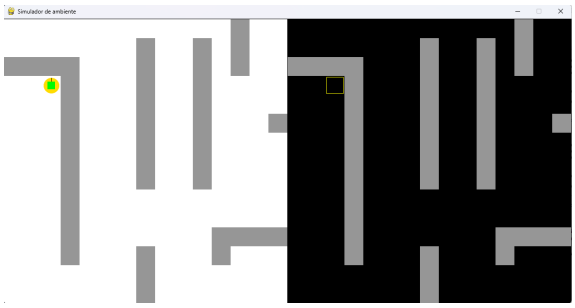


Figura 41: Após ter chegado ao objetivo final

Como se pode observar nestas imagens é possível verificar a utilidade de cada movimento que o agente pode realizar e a sua melhor politica.
Ao examinar os vários ambientes existentes foi possível observar uma diferença com o ambiente número 4 pelo valor do seu gama ser demasiado pequeno (0.85) para o processamento necessário neste ambiente. Ao aumentar o valor do gama para 0.95 já é possível realizar o processamento necessário para o agente atingir todos os objetivos.

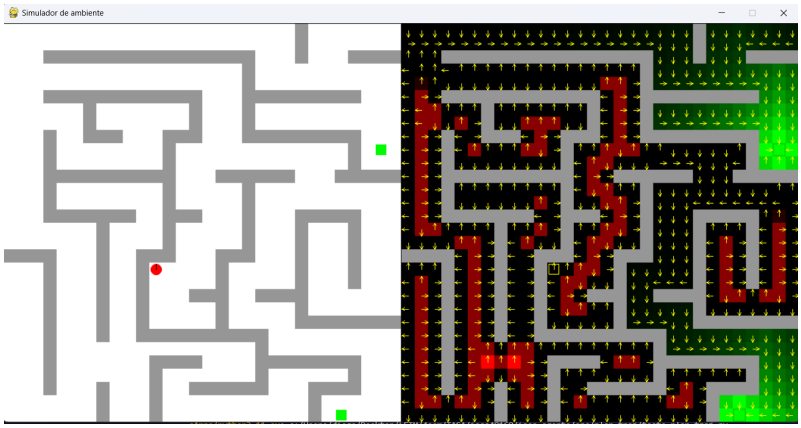


Figura 42: Ambiente 4 com gama=0.85

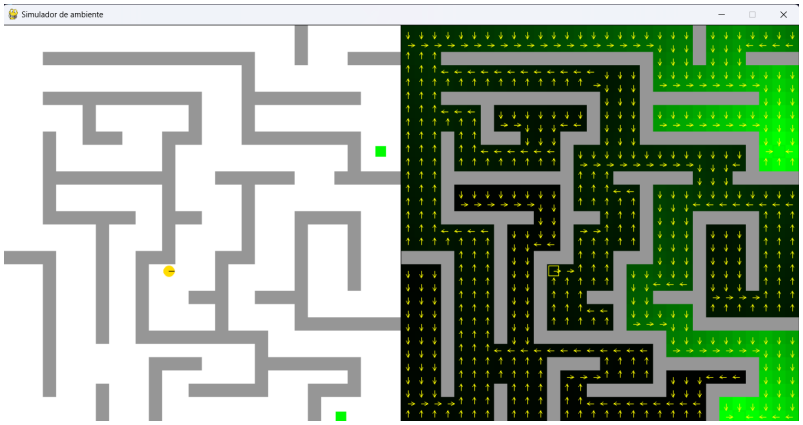


Figura 43: Ambiente 4 com gama=0.95

4 Revisão do Projeto Realizado

Ao longo da realização deste trabalho fui tendo alguns erros em certos métodos que me impossibilitavam de avançar no projeto por erros como não verificar a existência ou não existência de certo atributo, ou ao chamar certos métodos ou atributos não usar os nomes corretos ou não entender na altura que é privado ou protegido.

Um erro que tive foi no método **ciclo()** da classe "ProcuraProfLim"em que a alterar um nó em vez de igualar estava a fazer uma comparação entre dois diferentes, um erro que não é detetado mas que resulta no mau funcionamento do código.

Outro método onde obtive um erro foi no **obter-accao()** da classe "PlanoPEE"onde tinha a lógica do código errada. O suposto era verificar se a dimensão da dimensão era maior que 1 e se fosse *true* confirmar que o estado do primeiro nó é igual ao estado dado pelo método e se sim retornar o operador do primeiro nó que tivesse um operador ativo enquanto que no inicio apenas verificava se existia uma solução e retornava o operador do primeiro nó dessa solução, sendo que esse operador era inexistente.

5 Conclusão

Com o desenvolvimento contínuo deste projeto, foi possível o aprofundamento da matéria lecionada em aula tal como as arquiteturas de agentes inteligentes, reativos e deliberativos e o uso das classes UMLs.
Sinto que ao longo deste semestre fui capaz de aplicar a matéria dada em aula e que no futuro serei capaz de aplicar em outras cadeiras deste curso como em âmbito profissional principalmente as classes UMLs que é algo útil em qualquer área da informática principalmente da programação.

6 Bibliografia

Figuras

Imagem de Capa - https://exs.com.pt/o-que-ira-diferenciar-um-pt-personal-trainer-do-futuro-da-ia-inteligencia-artificial/

Figura 3 - https://en.wikipedia.org/wiki/Behavior_tree_(artificial_intelligence,_robotics_and_control)#

As restantes figuras foram retiradas ou dos slides fornecidos pelo docente ou resultados do código desenvolvido em aula.