RELATÓRIO: ENTREGA FINAL

PROJECTO CURRICULAR DE
INTELIGÊNCIA ARTIFICIAL PARA SISTEMAS AUTÓNOMOS

REALIZADO POR:
TATIANA DAMAYA (A50299)

DOCENTE:
CARLOS JÚNIOR

INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

LEIM-LEIC-23/24

ÍNDICE

INTRODUÇÃO

Este relatório, enquadra-se no âmbito da unidade curricular *Inteligência Artificial para Sistemas Autónomos*, da licenciatura de *Engenharia Informática e Multimédia*.
Esta unidade curricular, aborda essencialmente duas áreas da engenharia informática: Inteligência Artificial e Engenharia de Software.
Os projetos descritos no relatório, consistem num jogo e numa simulação de um agente. Estes projetos foram divididos em partes, sendo cada divisão proposta durante o decorrer das aulas, desenvolvida pelos alunos e submetida no fim destas.
O objetivo deste relatório é detalhar o processo do desenvolvimento dos projetos e tal como estes, o relatório também foi dividido, neste caso em duas entregas.
Esta consiste na entrega final, na qual abordarei não só o Projeto - Parte 1 e o Projeto - Parte 2, que já estavam na primeira entrega, como agora o Projeto - Parte 3 e Projeto - Parte 4, ficando a conter todas as partes realizadas.
A estrutura do relatório será composta, primeiramente por um enquadramento teórico da matéria necessária para a realização do projetos, e depois a explicação do código e procedimentos para as suas execuções.

04

ENQUADRAMENTO TEÓRICO

O que saber para realizar o Projecto – Parte 1, Projeto - Parte 2, Projeto - Parte 3 e Projeto - Parte 4?

INTELIGÊNCIA ARTIFICIAL:

Inteligência artificial (IA) é uma área que procura desenvolver sistemas capazes de realizar tarefas que normalmente exigem inteligência humana.
Dentro desse vasto domínio, existem diferentes paradigmas que orientam a abordagem e o desenvolvimento destes sistemas inteligentes.
Os principais paradigmas da IA são o simbólico, o conexionista e o comportamental, em que cada um oferece diferentes abordagens e técnicas para resolver problemas.

No paradigma simbólico, os sistemas são baseados em regras lógicas e representações simbólicas que manipulam símbolos para realizar tarefas de raciocínio, baseadas em conhecimento, procurando representar o conhecimento de especialistas num determinado campo. Esses sistemas, ganharam destaque na década de 1970, sendo capazes de tomar decisões ou fornecer assistência na tomada de decisões em domínios específicos, sem a necessidade de intervenção humana direta.

O paradigma conexionista, baseia-se em redes neurais artificiais, modeladas com base na estrutura e funcionamento do cérebro humano (sistema nervoso), especialmente dos neurónios e as suas conexões procurando simular a inteligência humana. Essas redes são capazes de aprender com dados, sendo amplamente utilizadas em tarefas de reconhecimento de padrões complexos, processamento de linguagem natural e visão computacional.

No paradigma comportamental, a inteligência emerge diretamente do comportamento de sistemas em interação com o ambiente, sugerindo que a inteligência pode ser compreendida e replicada através da análise e modelagem do comportamento observável em resposta a estímulos específicos, enfatizando a interação direta entre um agente e seu ambiente.

05 ENGENHARIA DE SOFTWARE:

À medida que os sistemas baseados em Inteligência Artificial se tornam cada vez mais complexos, a necessidade de métodos sólidos de Engenharia de Software torna-se crucial. Esta disciplina proporciona a estrutura necessária para lidar com a complexidade e a constante mudança nesse cenário, enfatizando a importância da modelagem adequada e da utilização de ferramentas como a Unified Modeling Language (UML).

Ao analisarmos a Engenharia de Software dois grandes desafios se destacam: a crescente complexidade dos sistemas e a necessidade contínua de adaptação às mudanças.

A mudança expressa no ritmo crescente a que o software necessita de ser produzido, ou modificado, para satisfazer as necessidades dos respetivos contextos de utilização, quer a nível de funcionalidades disponibilizadas, quer a nível das tecnologias utilizadas.

A complexidade de um sistema expressa-se numa dificuldade crescente em compreender e gerir as partes e as relações entre partes que constituem um sistema de software, a qual se pode observar numa representação gráfica (num modelo) desse software. Na prática, essa complexidade traduz-se na dificuldade e esforço crescente para a concepção e implementação do software, à medida que vão sendo incluídos mais aspectos do seu funcionamento. A complexidade, mede-se através de fatores como acoplamento, coesão, simplicidade e adaptabilidade, os quais influenciam a capacidade do sistema de evoluir e ser mantido de forma eficaz.

Para enfrentar essa complexidade, são essenciais abordagens de modelagem e organização eficazes, como a modularidade, a factorização e a abstração que ajudam a reduzir a interdependência entre os subsistemas, eliminar redundâncias e destacar apenas o essencial. Um sistema bem estruturado maximiza fatores como segurança, robustez e facilidade de manutenção, mantendo um baixo acoplamento e alta coesão entre seus módulos.

A Unified Modeling Language (UML) surge como uma ferramenta padronizada para modelar sistemas de software, facilitando a compreensão, comunicação e tradução de modelos em código. Através dos diferentes diagramas e elementos UML, é possível representar tanto a estrutura quanto o comportamento dos sistemas, fornecendo uma visão abrangente que auxilia no desenvolvimento e na manutenção do software.

Um aspecto crucial no processo de Engenharia de Software é o desenvolvimento iterativo, que encontra o equilíbrio ideal entre a modularidade do projeto e sua complexidade. Isso envolve saber até que ponto decompor as funcionalidades, garantindo que o software seja robusto e bem documentado ao longo do processo de desenvolvimento.

06

AGENTES:

O modelo geral de um agente inteligente incorpora o ciclo de:
percepção-processamento-ação

Agentes inteligentes são caracterizados por qualidades fundamentais para que alcancem os seus objetivos de maneira eficaz, estas são: autonomia (capacidade de um sistema operar por si próprio, de modo independente de outros sistemas), reactividade (capacidade de um sistema reagir aos estímulos do ambiente), pró-actividade (capacidade de um sistema tomar a iniciativa de acção em função dos seus objectivos), sociabilidade (capacidade de um sistema inter actuar e agir em conjunto com outros agentes para concretizar objectivos individuais ou comuns a outros agentes).
Um exemplo prático, pode ser observado em personagens virtuais de jogos de computador, que são modelados como agentes autónomos. Estes personagens são projetados para perceber condições no ambiente de jogo, processar essas informações e, em seguida, agir de acordo com estratégias predefinidas para alcançar objetivos dentro do jogo.

Dentro deste contexto, as arquiteturas de agentes inteligentes podem ser classificadas em três categorias principais, onde cada tipo define a organização interna e o método pelo qual um agente executa as suas funções, influenciando diretamente como ele percebe e interage com o ambiente. As categorias são: Reativa (Paradigma Comportamental) o comportamento do sistemas é gerado de forma reactiva, com base em associações entre estímulos (referentes às percepç ões) e respostas (referentes às acções).
Deliberativa (paradigma simbólico), o comportamento do sistema é gerado com base em mecanismos de deliberação, raciocínio e tomada de decisão, utilizando representações internas que incluem a representação explícita de objectivos, e -
Híbrida, projetada para combinar as vantagens das arquiteturas deliberativa e reativa, permitindo que os agentes tenham um desempenho mais eficiente e adaptativo em ambientes complexos. Essa integração tem como objetivo harmonizar a capacidade de tomar decisões, baseadas em raciocínio e planeamento detalhado (características da arquitetura deliberativa), com a habilidade de responder de forma rápida e direta aos estímulos do ambiente (características da arquitetura reativa).

07

AMBIENTE:

O ambiente, em que os agentes inteligentes operam, desempenha um papel crucial na inteligência artificial, pois fornece o contexto e os desafios que moldam a atuação do agente. Esse ambiente, pode ser tanto físico quanto virtual, cada um com as suas características específicas e desafios associados.

Ambientes Físicos: Aqui, os agentes usam sensores e atuadores reais para interagir com o mundo, como robôs em fábricas ou veículos autónomos. Eles colectam dados através de sensores e atuam fisicamente para realizar tarefas ou resolver problemas.

Ambientes Virtuais: Em espaços digitais, como jogos ou simulações, os agentes processam dados digitais e executam ações dentro do ambiente simulado. Esses ambientes permitem testar e treinar IA em condições controladas, sem os riscos do mundo físico.

08 ARQUITECTURA DE AGENTES:

Arquiteturas de Agentes (Componentes Básicos de um Agente):

- Agente: Entidade que interage com o ambiente.
- Ambiente: Contexto no qual o agente opera.
- Percepção: Informações que o agente obtém do ambiente.
- Ação: Respostas do agente ao ambiente.
- Reação: Comportamentos imediatos do agente a estímulos.

Arquitetura Reativa - o processamento interno é baseado em reações diretas a estímulos presentes nas percepções. Essas reações produzem respostas que geram ações, sem manter um estado ou memória de situações passadas, resultando em soluções rápidas, mas geralmente subótimas.
Exemplo de Arquitetura Reativa:

- Comportamentos de Exploração: Exploração baseada na aleatoriedade.

Arquitetura Reativa com Memória -Nesta variante, as reações podem manter estado ou memória de situações passadas, permitindo, por exemplo, evitar situações desfavoráveis anteriores.
Exemplo de Arquitetura Reativa com Memória:

- Comportamentos de Exploração: Exploração com base na memória de posições anteriores, sem suporte para planeamento otimizado de comportamentos.

Tempo e Comportamento

- Presente:
  - Agentes reativos sem estado: Capazes apenas de reagir.
- Passado e Presente:
  - Agentes reativos com estado: Capazes de repetir ou evitar situações passadas.
- Passado, Presente e Futuro:
  - Agentes deliberativos: Capazes de antecipar futuros estados, otimizando o comportamento presente para atingir objetivos futuros.

09 ARQUITECTURA DE AGENTES REATIVOS:

Na arquitetura de agentes reativos o comportamento do sistema define um ciclo percepção-reacção-acção, onde as reacções definem de forma modular as associações entre estímulos (derivados da percepção) e respostas (geradoras de acção), ou seja, fundamenta-se na relação direta entre estímulos percebidos e respostas geradas, estas ações do agente são ativadas diretamente, sem a necessidade de representações internas complexas do ambiente, por exemplo simbolicamente. No contexto desse modelo, as percepções podem desencadear diversas reações, resultando em múltiplas ações possíveis. Para determinar a ação mais apropriada, são empregados mecanismos como execução paralela, priorização de ações, combinação de ações e coordenação de comportamentos.

Mecanismos de Seleção de Ações:
- Execução Paralela: Ações não interferentes podem ser executadas simultaneamente, desde que a infraestrutura do agente suporte múltiplos atuadores.
- Priorização de Ações: Quando há interferência entre as ações, a seleção é feita com base na prioridade de cada uma, prevalecendo a ação de maior prioridade.
- Combinação de Ações: As ações podem ser combinadas numa única ação composta, frequentemente representada como vetores somáveis para formar uma nova ação.

Coordenação de Comportamentos:
- Hierarquia de Subsunção: Os comportamentos são organizados hierarquicamente, onde os de nível superior podem suprimir ou substituir os de nível inferior.
- Seleção por Prioridade: A resposta selecionada é aquela com maior prioridade no contexto atual.
- Composição de Respostas: As respostas individuais são combinadas numa única ação composta, como a soma vetorial.
- Modelação de Comportamentos: Os comportamentos dos agentes são modelados com base em uma análise detalhada dos objetivos, subdivididos em comportamentos e sub-comportamentos necessários para alcançá-los.

Os comportamentos são conjuntos de reações, subdivididos em comportamentos e sub-comportamentos que estão relacionados entre si, procurando alcançar um resultado específico. Por exemplo, um comportamento pode ser responsável por evitar obstáculos, enquanto outro pode ser responsável por se aproximar de alvos. Essa modularização permite uma organização mais clara e eficiente das ações do agente.

Um comportamento relaciona padrões de percepção com padrões de ação. No caso geral, um comportamento pode ser composto por outros comportamentos
Comportamento Composto: Agrega conjuntos de comportamentos individuais, permitindo lidar com situações mais complexas ao combinar diferentes comportamentos num único conjunto coerente de ações.

Exemplo Prático: Agente Prospector: Este agente é projetado para navegar autonomamente num ambiente com obstáculos, recolhendo alvos. Os sub-comportamentos incluem aproximar-se de alvos, evitar obstáculos e explorar o ambiente. A seleção de ação é baseada na proximidade dos alvos, priorizando movimentos na direção de alvos mais próximos.

Este quadro teórico abrange não apenas a execução de tarefas em ambientes controlados, mas também se estende à robótica autônoma e sistemas multi-agentes, onde a tomada de decisão rápida e precisa é crítica.

10 RACIOCÍNIO AUTOMÁTICO E TOMADA DE DECISÃO

Resolução Automática de Problemas - Área da inteligência artificial focada na resolução de problemas através de meios algorítmicos, utilizando raciocínio automático. Este tipo de raciocínio permite que um sistema computacional resolva problemas com base em representações de conhecimento do domínio específico, gerando soluções a partir de diversas alternativas.

O raciocínio automático envolve dois tipos principais de atividades:

- 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: Ganhos ou perdas, medidos em termos de utilidade.

Estrutura e Dinâmica

- Estados: Representam configurações possíveis de um problema ou sistema.
- Operadores: Representam transformações que podem ocorrer no estado de um problema ou sistema.
- Espaço de Estados: Conjunto de estados e de transições de estado, representado como um grafo.
- Problema: Combinação do estado inicial, operadores disponíveis e objetivo final.
- Mecanismo de Raciocínio: Exploração de opções possíveis para encontrar uma solução através de simulação prospetiva.

Exemplo de Aplicações

- Problemas matemáticos
- Jogos diversos
- Navegação autónoma
- Planeamento robótico
- Planeamento de sistemas logísticos
- Controlo de personagens multimédia

A resolução de problemas através de raciocínio automático requer a conversão de informação concreta do domínio do problema em estruturas simbólicas internas. Esta conversão é realizada por processos de codificação (transformação de informação concreta em simbólica) e descodificação (transformação de simbólica em concreta, por exemplo, ações a realizar).

Para encontrar soluções, são utilizados vários métodos de procura:

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

Variantes da Procura Melhor-Primeiro

- Procura de Custo Uniforme: Minimização do custo acumulado até cada nó explorado.
- Procura Sofrega: Minimização da estimativa de custo para atingir o objetivo, sem considerar o custo do percurso explorado, resultando em uma solução sub-ótima.
- Procura A*: Minimização do custo global usando uma heurística admissível.
  - h(n): Estimativa de custo (heurística) para atingir o estado objetivo a partir do estado inicial, utilizada nas procuras Sofrega e A*.

Exemplo: Mundo dos Blocos

No mundo dos blocos, o objetivo é controlar um braço robótico para mover blocos sobre uma mesa, atingindo uma configuração específica. A solução é encontrada através da simulação das ações possíveis e da exploração de diferentes configurações.

11 ARQUITETURA DE AGENTES DELIBERATIVOS

Na arquitetura deliberativa, a memória desempenha um papel central na geração do comportamento do agente. Esta memória suporta a representação do mundo e os mecanismos de deliberação, incluindo raciocínio e tomada de decisão.

Componentes da Arquitetura Deliberativa:

- Representação Interna do Problema (Modelo do Mundo): Utilizada para simular e explorar diferentes opções.
- Raciocínio sobre Fins (Deliberação): Decidir o que fazer.
- Raciocínio sobre Meios (Planeamento): Decidir como fazer.
- Planos de Ação: Definidos com base em representações de objetivos, ações realizáveis e o ambiente.

Raciocínio Prático - No contexto de um agente autônomo, o raciocínio prático é orientado para a ação, utilizando representações simbólicas dos objetivos, ações e do ambiente para gerar planos de ação.

Processo Geral de Tomada de Decisão e Ação

1. Observar o Mundo: Geração de percepções.
2. Atualizar o Modelo do Mundo: Baseado nas percepções.
3. Deliberar o Que Fazer: Gerar objetivos.
4. Planejar Como Fazer: Criar um plano de ação.
5. Executar o Plano de Ação: Implementação das ações decididas.

Problemas no Raciocínio Prático

- Recursos Computacionais Limitados: Memória e tempo de computação.
- Dinamismo do Ambiente: Mudanças no ambiente podem tornar o raciocínio desatualizado.
- Reconsideração de Opções: Necessidade de reavaliar planos devido a mudanças no ambiente.

Tipos de Processos Deliberativos:

- Planeamento Automático - Tem como objetivo gerar sequências de ação (planos) para concretização de objetivos pré-definidos, utilizando 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

- Planeamento Baseado em PDM - Este planeamento utiliza a representação do problema como um processo de decisão de Markov (PDM), produzindo uma política de ação que define a ação a ser realizada para cada estado possível.

Utilidade e Política

O raciocínio automático com base em PDM envolve o cálculo da função de utilidade (valor), que define o valor associado a cada estado do problema. Essa função suporta o cálculo da política de ação, determinando a seleção de ações que maximizam o valor em cada estado.

12 APRENDIZAGEM POR REFORÇO

Problemas de Decisão Sequencial

- Incerteza **-** Nos problemas de decisão sequencial, o valor das ações de um agente (ganhos e perdas) não depende de decisões simples baseadas apenas no estado atual, mas de uma sequência de ações encadeadas no tempo. Os resultados das ações podem ser incertos, ou seja, não totalmente controlados (não determinísticos).
- Recompensas Diferidas - As recompensas ocorrem ao longo do tempo e dependem de uma série de ações tomadas em sequência.

Tomada de Decisão com Incerteza
A incerteza resulta da impossibilidade de obter informação completa relativa ao domínio do problema. Exemplo: Navegação em veículos autónomos.

Processos de Decisão de Markov (PDM)
Os PDM são uma representação do mundo, que inclui:
- S: Conjunto de estados do mundo.
- A(s): Conjunto de ações possíveis enum estado.
- T(s, a, s'): Probabilidade de transição de s para s' através de a.
- R(s, a, s'): Recompensa esperada na transição de s para s' através de a.
- t = 0, 1, 2, ...: Tempo discreto.

Problemas enfrentados:
- Dimensão dos espaços de estados (problema da dimensionalidade).
- Dificuldade de definição de modelos do mundo (por exemplo, a partir de dados experimentais).
- Modelos de transição T(s, a, s') e de recompensa R(s, a, s') desconhecidos.

13 APRENDIZAGEM POR REFORÇO

Aprendizagem por Reforço

A aprendizagem por reforço envolve aprendizagem incremental a partir da experiência. Um agente aprende através da interação com o ambiente, recebendo recompensas ou punições pelas suas ações.
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.

Componentes:
- Estado
- Ação
- Reforço (Ganho/Perda)

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).

Aplicações:
- Robótica
- Jogos (treinar NPCs para tomar melhores decisões)
- Finanças
- Condução autônoma

Processo de Aprendizagem
1. Observar o estado (s).
2. Executar uma ação (a).
3. Receber uma recompensa (r).
4. Observar o novo estado (s').
5. Escolher a próxima ação (a').

Aprendizagem Automática
Para melhorar o desempenho numa dada tarefa (T), com base numa medida de desempenho (D) e na experiência (E).

Exemplo: Aprender a jogar xadrez:
- T: Jogar xadrez.
- D: Percentagem de jogos ganhos.
- E: Jogos realizados.

Aprendizagem ≠ Memorização
A aprendizagem envolve generalização e formação de abstrações (modelos).a.

PROJETO - PARTE 1 (JOGO)

OBJETIVO:

Pretende-se implementar um jogo com uma personagem virtual que interage com um jogador humano. O jogo consiste num ambiente onde a personagem tem por objectivo registar a presença de animais através de fotografias. Quando o jogo se inicia a personagem fica numa situação de procura de animais. Quando detecta algum ruído aproxima-se e fica em inspecção da zona, procurando a fonte do ruído. Quando volta a haver silêncio a personagem volta a uma situação de procura de animais. Quando detecta um animal a personagem aproxima-se e fica em observação. Caso o animal continue presente, a personagem observa o animal e fica preparada para o registo, se ocorrer a fuga do animal a personagem fica em inspecção da zona, à procura de uma fonte de ruído. Na situação de registo, se o animal continuar presente fotografa-o, caso ocorra a fuga do animal ou a personagem tenha conseguido uma fotografia do animal, a personagem fica novamente numa situação de procura. A interacção com o jogador é realizada em modo de texto.

PROCESSO:

Com o conhecimento previamente abordado, ao ler o enunciado da primeira parte do projeto conseguimos perceber que precisamos de criar um ambiente virtual. E nesse ambiente existe uma personagem virtual (agente virtual) que interage com um jogador humano.
O comportamento do sistema corresponde à forma como o sistema age, ou seja, activa as suas saídas (gera informação de saída), em função das suas entradas (informação de entrada, proveniente do exterior) e do seu estado interno. Um sistema computacional geral, pode assim ser caracterizado de forma abstrata, através de uma representação de estado, que define em cada momento a configuração interna do sistema, e de uma função de transformação que gera as saídas e o próximo estado em função das entradas e do estado actual do sistema.
Entradas e saídas abstraídas serão realizadas em termos dos conjuntos de símbolos que nelas podem ocorrer – Esses conjuntos de símbolos são designados alfabetos – Consideremos um alfabeto de entrada Σ = { Silencio, Ruido, Animal, Fuga, Fotografia, Terminar } que saão os eventos do ambiente, um alfabeto de saída Z = { Procurar, Aproximar, Observar, Fotografar } que são as ações da personagem e um conjunto de estados que caracterizam a personagem Q = { Procura, Inspecção, Observação, Registo }. Estas funções serão posteriormente convertidas numa máquina de estados.

Função de transição de estado
$\delta: Q \times \Sigma \rightarrow Q$

Estado	Evento	Novo estado
Procura	Animal	Observação
Procura	Ruido	Inspecção
Procura	Silencio	Procura
Inspecção	Animal	Observação
Inspecção	Ruido	Inspecção
Inspecção	Silencio	Procura
Observação	Fuga	Inspecção
Observação	Animal	Registo
Registo	Animal	Registo
Registo	Fuga	Procura
Registo	Fotografia	Procura

Função de saída
$\delta: Q \times \Sigma \rightarrow Z$

Estado	Evento	Acção
Procura	Animal	Aproximar
Procura	Ruido	Aproximar
Procura	Silencio	Procurar
Inspecção	Animal	Aproximar
Inspecção	Ruido	Procurar
Inspecção	Silencio	
Observação	Fuga	
Observação	Animal	Observar
Registo	Animal	Fotografar
Registo	Fuga	
Registo	Fotografia	

ESTRUTURA

Para este projeto usei o IDE eclipse e a linguagem de programação Java.
Para começar o projecto – parte 1 meti as pastas de acordo com o pedido pelos professores com a estrutura:

iasannnnn
└ iasa_jogo
└ src
└ ...

Depois ao consultar os slides da ARQUITECTURA DE APOIO AO PROJECTO para organizar melhor o código, garanti que as classes estariam todas bem organizadas consultando os UMLs primeiro, criando todos os packages necessários: agente , ambiente,jogo (com sub-packages: ambiente e personagem) e maqest.
Depois slide a slide fui criando as classese métodos necessários antes de perceber e realizar os seus procedimentos.

PACKAGE AMBIENTE:

O modelo de ambiente, considera um ambiente que evolui no tempo, onde é possível executar comandos e observar eventos, e onde é possível mostrar esses mesmos comandos e eventos

INTERFACES • Contractos funcionais para interação com o exterior

Dentro deste package temos 3 interfaces Ambiente, Evento, Comando, todas estas classes têm métodos de visibilidade publica.

Ambiente: Define as operações básicas que um ambiente deve suportar para ser utilizado por um agente. As operações incluem a evolução do estado do ambiente, observação do estado atual para identificar eventos e execução de comandos.

Evento: Representa um evento que pode ocorrer dentro do ambiente, como a detecção de um ruído, etc.

Comando: Representa um comando que pode ser executado no ambiente. Um comando pode ser qualquer ação que afeta o estado do ambiente de alguma forma, como movimento de um personagem, interação com objetos, etc.

PACKAGE AMBIENTE:

Interface AMBIENTE: A interface Ambiente, define um contrato para as classes que implementam estas funcionalidades, garantindo que qualquer implementação do ambiente no jogo irá fornecer mecanismos para evoluir o estado do ambiente, observar eventos, e responder a comandos. Esta estrutura é essencial para a dinâmica do jogo, onde a interação contínua e a resposta ao ambiente são cruciais para uma experiência de jogo envolvente e interativa.

- public void evoluir();
  - Este método permite que o ambiente onde o jogo se desenrola pode mudar ou evoluir com o tempo. A função evoluir é responsável por evoluir o estado do ambiente ao longo do jogo, observar o estado atual e executar um comando no ambiente.
- public Evento observar();
  - O método observar permite que o ambiente seja inspecionado para detectar eventos. A função retorna um objeto Evento, que encapsula informações sobre o que foi observado no ambiente.
- public void executar(Comando comando);
  - O método executar permite que o ambiente responda a comandos ou ações enviadas, para fazer a personagem interagir com o ambiente.

Interface COMANDO: representa uma ação ou comando que pode ser emitido dentro do ambiente do jogo.

- public void mostrar();
  - serve para representar o comando de forma textual no jogo, permitindo uma forma de feedback ou debug para entender o que está sendo comandado no ambiente.

Interface EVENTO: define um evento que ocorre no ambiente do jogo.

- public void mostrar();
  - usado para representar o evento de forma textual, para informar o jogador sobre o que está a acontecer no jogo ou novamente, ser útil para debug durante o desenvolvimento.

Esses componentes são essenciais para a criação de um sistema interativo em que a personagem e o ambiente respondem dinamicamente às entradas do jogador e a eventos automáticos gerados pelo jogo.

PACKAGE AGENTE:

Classe Accao: encapsula um Comando, que é uma ação a ser executada no ambiente do jogo. A classe serve como um intermediário entre as decisões tomadas pelo controle do agente e as ações executadas no ambiente.

- public Accao(Comando comando):
  - Construtor: Aceita um Comando como parâmetro e o associa à instância da classe Accao. Este comando é o que será executado pelo agente no ambiente.

Classe Abstrata Agente: define o comportamento base para todos os agentes no jogo. Cada agente possui um ambiente em que opera e um controle que define como ele responde às percepções do ambiente.

- public Agente(Ambiente ambiente, Controlo controlo):
  - Inicializa o agente com um ambiente específico e um sistema de controlo. Estes são usados respectivamente para observar eventos e decidir ações.
- public void executar():
  - Executa um ciclo completo de percepção, processamento e ação. O agente primeiro percepciona o ambiente, depois processa essa percepção para determinar uma ação, e finalmente atua no ambiente conforme a ação decidida.
- protected Percepcao percepcionar():
  - Observa o ambiente para detectar eventos e cria uma Percepcao baseada no evento observado. Esta percepção é então usada para tomar decisões.
- protected void actuar(Accao accao):
  - Recebe uma Accao e, se válida, executa o comando associado no ambiente. Este método é essencial para a interação do agente com o ambiente.

Interface Controlo: define um contrato para sistemas de controle que processam percepções e determinam ações. Isso permite a implementação de diferentes lógicas de decisão para diferentes tipos de agentes.

- public Accao processar(Percepcao percepcao):
  - Recebe uma Percepcao do ambiente e retorna uma Accao baseada nesta percepção. O método é onde a lógica de decisão do agente é implementada, convertendo eventos do ambiente em comandos de ação.

Classe Percepcao: serve como uma representação de um evento observado no ambiente. Esta classe permite encapsular informações sobre eventos de forma que possam ser utilizadas pelos sistemas de controlo para tomar decisões.

- public Percepcao(Evento evento):
  - Aceita um Evento como parâmetro e associa este evento à instância da classe Percepcao.
- public Evento getEvento():
  - Retorna o evento associado a esta percepção. Este método permite que o controlo do agente acesse aos detalhes do evento para tomar as suas decisões.

Cada um destes componentes é projetado para interagir de maneira coesa dentro da arquitetura do sistema, facilitando a modularidade e a flexibilidade na implementação de comportamentos de agentes no jogo.

PACKAGE MAQEST:

Classe Estado: representa um estado dentro da máquina de estados. Ela armazena transições que definem para qual outro estado a máquina deve mudar dado um evento específico, e a ação que deve ser executada durante essa transição.

- public Estado(String nome):
  - Inicializa um estado com um nome e prepara um mapa para armazenar as transições possíveis.
- public Transicao processar(Evento evento):
  - Dado um evento, procura no mapa de transições e retorna a transição correspondente. Se nenhum evento corresponder, retorna null.
- public Estado transicao(Evento evento, Estado estadoSucessor):
  - Define uma transição para um estado sucessor dado um evento.
- public Estado transicao(Evento evento, Estado estadoSucessor, Accao accao):
  - Similar ao método anterior, mas permite definir uma ação a ser executada durante a transição.

Classe MaquinaEstados : gere o estado atual da máquina de estados e processa eventos para mover-se entre os estados conforme definido pelas transições.

- public MaquinaEstados(Estado estadoInicial):
  - Inicia a máquina de estados com um estado inicial.
- public Accao processar(Evento evento):
  - Processa um evento, encontrando a transição correspondente no estado atual e, se uma transição for encontrada, muda o estado para o estado sucessor e retorna a ação associada à transição. Se nenhuma transição for encontrada, retorna null.

Classe Transicao: encapsula a informação sobre uma transição específica dentro da máquina de estados, incluindo o estado para o qual mover-se e a ação a ser executada durante a transição.

- public Transicao(Estado estadoSucessor, Accao accao):
  - Construtor: Inicializa uma transição com um estado sucessor e uma ação opcional.
- public Estado getEstadoSucessor():
  - Obter Estado Sucessor: Retorna o estado para o qual esta transição leva.
- public Accao getAccao():
  - Obter Ação: Retorna a ação associada a esta transição.

Essas classes juntas permitem uma modelagem flexível de comportamentos de entidades dentro do jogo, controlando como eles respondem a eventos do ambiente e mudam o seu estado interno e ações de forma coerente e previsível.

PACKAGE JOGO:

JOGO.AMBIENTE:

Classe AmbienteJogo: Implementa a interface Ambiente para criar um ambiente de jogo interativo que gera eventos baseados em entradas do usuário e responde a comandos.

- public AmbienteJogo():
  - Inicializa o mapa de eventos com diferentes teclas correspondendo a diferentes eventos do jogo.
- @Override public void evoluir():
  - Sobrescreve o método da interface Ambiente. Gera um novo evento baseado no input que o utilizador pôs na consola.
- @Override public Evento observar():
  - Sobrescreve o método da interface Ambiente. Exibe o evento atual e o retorna.
- @Override public void executar(Comando comando):
  - Sobrescreve o método da interface Ambiente. Executa e mostra um comando fornecido.
- private EventoJogo gerarEvento()
  - Solicita ao utilizador a entrada de um evento para ser gerado e retorna silêncio como padrão se o evento não for reconhecido.

Enum ComandoJogo: Implementa a interface Comando para definir os comandos possíveis no jogo: PROCURAR, APROXIMAR, OBSERVAR, FOTOGRAFAR;

- @Override public void mostrar():
  - Sobrescreve o método da interface Comando. Exibe o comando que está sendo executado.

Enum EventoJogo: Implementa a interface Evento para definir os tipos de eventos que podem ocorrer no jogo: SILENCIO, RUIDO, ANIMAL, FUGA, FOTOGRAFIA, TERMINAR;

- @Override public void mostrar():
  - Sobrescreve o método da interface Evento. Exibe o evento que ocorreu.

JOGO.PERSONAGEM:

Classe ControloPersonagem: Implementa a interface Controlo para gerir as decisões do personagem com base nos eventos percebidos, utilizando uma máquina de estados para determinar ações.

- public ControloPersonagem():
  - Configura os estados e transições da máquina de estados.
- @Override public Accao processar(Percepcao percepcao):
  - Sobrescreve o método da interface Controlo. Processa o evento percebido para determinar a ação subsequente e mostra o estado atual da máquina de estados.
- private void mostrar():
  - Exibe o estado atual da máquina de estados.

Classe Personagem: Estende a classe Agente, configurando o ambiente e o controlo específico para o personagem dentro do jogo.

- public Personagem(AmbienteJogo ambiente):
  - Configura o ambiente e o controlo do personagem.

PACKAGE JOGO:

CLASSE JOGO:

Classe Principal (jogo.Jogo): Contém o método main para iniciar e controlar o fluxo do jogo, gere as interações entre o ambiente e o personagem até que o evento de terminação seja gerado.

- private static void executar():
  - Executar Jogo: Executa um ciclo de evolução do ambiente e ação do personagem até o fim do jogo.
- public static void main(String[] args):
  - Ponto de Entrada do Programa: Inicializa o ambiente e o personagem e começa a execução do jogo.

1. O jogo inicia com a personagem em estado de procura de animais.
2. Quando detecta ruído, a personagem se aproxima e entra em inspeção da zona.
3. Após detectar um animal, a personagem entra em estado de observação.
4. Se o animal permanecer presente, a personagem fica preparada para o registro.
5. Em caso de fuga do animal, a personagem volta para inspeção da zona.
6. Durante o registro, se o animal permanecer, a personagem fotografa-o.
7. Após a fuga do animal ou a obtenção da fotografia, a personagem retorna ao estado de procura.
8. A interação com o jogador é realizada em modo de texto, permitindo a entrada de comandos para controlar a personagem e interagir com o ambiente.

EXECUÇÃO DO CÓDIGO:

O ambiente e o personagem são inicializados e começa a execução do jogo. Nesta execução o ambiente vai estar constantemente a evoluir, ou seja a gerar um evento (comando) que é introduzido pelo jogador humano, e o personagem a executar o ciclo (percepcionar (observar o ambiente para saber qual o evento ocorrido) -> processar (ver na máquina de estados qual é a ação para aquele evento, que primeiro percebe qual a transição associada a esse evento se houver, ao sabermos qual a transição podemos ver o seu estado sucessor, e retornar a ação para esta transição, depois faz print da ação) -> ação (executar o comando dessa acção, ou seja mostrar, fazer print do estado em que o jogo ficou)), isto enquanto o evento do ambiente/EventoJogo não for TERMINAR.

21

PROJETO - PARTE 1 (JOGO)

EXECUÇÃO DO PROGRAMA

```
Introduzir um evento:

s -> SILENCIO, r -> RUIDO, a -> ANIMAL
f -> FUGA, p -> FOTOGRAFIA, t -> TERMINAR
r
Evento: RUIDO
Estado: INSPEÇÃO
Ação: APROXIMAR


Introduzir um evento:

s -> SILENCIO, r -> RUIDO, a -> ANIMAL
f -> FUGA, p -> FOTOGRAFIA, t -> TERMINAR
s
Evento: SILENCIO
Estado: PROCURA

Introduzir um evento:

s -> SILENCIO, r -> RUIDO, a -> ANIMAL
f -> FUGA, p -> FOTOGRAFIA, t -> TERMINAR
a
Evento: ANIMAL
Estado: OBSERVAÇÃO
Ação: APROXIMAR


Introduzir um evento:

s -> SILENCIO, r -> RUIDO, a -> ANIMAL
f -> FUGA, p -> FOTOGRAFIA, t -> TERMINAR
f
Evento: FUGA
Estado: INSPEÇÃO

Introduzir um evento:

s -> SILENCIO, r -> RUIDO, a -> ANIMAL
f -> FUGA, p -> FOTOGRAFIA, t -> TERMINAR
o
Evento: SILENCIO
Estado: PROCURA

Introduzir um evento:

s -> SILENCIO, r -> RUIDO, a -> ANIMAL
f -> FUGA, p -> FOTOGRAFIA, t -> TERMINAR
t
Evento: TERMINAR
Estado: PROCURA
```

PROJETO - PARTE 2 (AGENTE PROSPECTOR)

OBJETIVO:

Objectivo: Realização de um sistema autónomo inteligente capaz de navegar num espaço de dimensões discretas, com obstáculos e um alvo, desviando-se dos obstáculos e recolhendo os alvos.

Direcções de movimento e de percepção do sistema: • Norte • Sul • Este • Oeste

AGENTE PROSPECTOR
• OBJECTIVOS
– Recolher alvos
•SUB-OBJECTIVOS
–Aproximar alvo:
• Aproximar alvo (direcção = NORTE)
• Aproximar alvo (direcção = SUL)
• Aproximar alvo (direcção = ESTE)
• Aproximar alvo (direcção = OESTE)
–Evitar obstáculos
• Evitar direccional nas 4 direcções
–Explorar

PROCESSO:

Para começar o projecto – parte 2, na primeira aula, tivemos um acompanhamento do professor na realização da estrutura correta para inicializá-lo. Isto foi feito com a adição da pasta do Simulador de Ambiente de Execução (SAE) disponibilizada no moodle e com o ficheiro .env e os imports necessários para a sua execução. Infelizmente o meu IDE ou sistema operativo estava a dar bastantes problemas com os imports do sae e soluções para tais, o que me levou a iniciar o trabalho sem certezas de que estaria a funcionar na sua totalidade. Após várias pesquisas e tentativas foi encontrada uma solução, importando os módulos necessários, como sys e Path da biblioteca pathlib, que são usados para manipulação de caminhos, para tal no fim da realização do tal, o que permitiu que todos os objetivos fossem alcançados.
Para a execução do código, consultei os slides da ARQUITECTURA DE APOIO AO PROJECTO para garantir que todas as classes estariam bem organizadas, também consultei os slides sobre a arquitetura de agentes reativos, onde obtive algumas explicações sobre o package que tinhamos de criar chamada ECR. Este package ECR e o SAE estram ambos numa pasta chamada lib tal como proposto pelo professor.

PACKAGE SAE:

O simulador de ambiente de execução (SAE) é uma plataforma que permite a execução de um agente autônomo num ambiente bidimensional simulado, composto por alvos e obstáculos.
O modelo de agente implementado pelo SAE segue uma arquitetura específica, onde a execução do agente ocorre ciclicamente, processando a cada passo de execução.
A arquitetura de agente segue um ciclo de percepção, processamento e ação, onde o agente processa sua percepção do ambiente e toma decisões com base nela.
A plataforma SAE oferece um conjunto de classes para desenvolvimento e teste de arquiteturas de agentes autônomos. Destacam-se as classes Simulador, Agente e Controlo, que coordenam a simulação e definem a base para a implementação de agentes específicos.
Para utilizar a plataforma SAE, é necessário iniciar uma instância da classe Simulador, indicando o número do ambiente e a instância do agente a ser executada.

Ambiente Gráfico:
A plataforma SAE apresenta um ambiente gráfico bidimensional, composto por duas áreas de visualização: uma área mostra o ambiente com o agente e sua direção de movimento, enquanto a outra área exibe informações internas do agente.

Comandos do Simulador:
Durante a execução, o simulador suporta os seguintes comandos de teclado:

- t: Terminar a simulação
- i: Iniciar o ambiente
- p: Ativar o modo de pausa (execução passo-a-passo)
- e: Executar um passo
- v: Alternar entre velocidade máxima e normal

Pré-Requisitos de Software:
Antes de utilizar a plataforma SAE, é necessário garantir que o sistema atenda aos seguintes requisitos de software:

- Python 3.11 ou superior
- Pygame

PACKAGE ECR:

Reaccao – Implementação do mecanismo base de uma reacção.

- def __init__(self, estimulo: Estimulo, resposta: Resposta):
  - Este método é o construtor da classe Reaccao.
  - Ele recebe um objeto estimulo do tipo Estimulo e um objeto resposta do tipo Resposta.
  - Estes objetos representam a detecção de um estímulo presente numa percepção e a geração de uma resposta a esse estímulo.

- def activar(self, percepcao: Percepcao) -> Accao:
  - Este método recebe como entrada um objeto percepcao do tipo Percepcao, que representa a percepção atual do ambiente.
  - O método utiliza o objeto estimulo associado à reação para detectar a intensidade do estímulo na percepção atual.
  - Se a intensidade do estímulo for maior que zero, o método ativa a resposta associada ao estímulo, passando a percepção atual e a intensidade do estímulo como parâmetros.
  - A ação resultante da resposta é retornada.
  - Se nenhum estímulo for detectado (ou seja, a intensidade for igual a zero), o método retorna None.

Comportamento – Interface que define a funcionalidade geral de um comportamento

- class Comportamento(ABC):
  - A classe Comportamento é definida como uma classe abstrata (ABC), o que significa que ela não pode ser instanciada diretamente. Em vez disso, outras classes devem herdar dela e fornecer implementações para seus métodos abstratos.

- @abstractmethod
- def activar(self, percepcao: Percepcao) -> Accao:
  - Este método é marcado como abstrato usando o @abstractmethod. Isso significa que qualquer classe que herde Comportamento deve fornecer uma implementação para este método.
  - O método activar recebe um parâmetro percepcao do tipo Percepcao, que representa a percepção atual do agente sobre o ambiente.
  - O método retorna um objeto do tipo Accao, que representa a ação a ser executada pelo agente com base em sua percepção.
  - Não há implementação real neste método, apenas a declaração pass, indicando que a implementação deve ser fornecida nas subclasses.

PACKAGE ECR:

ComportComp – Implementação do mecanismo base de um comportamento composto

- class ComportComp(Comportamento):
  - A classe ComportComp é definida como uma subclasse de Comportamento, o que significa que ela herda as propriedades e métodos da classe Comportamento.

- def __init__(self, comportamentos: list[Comportamento]):
  - Este método é o construtor da classe e é responsável por inicializar o comportamento composto com uma lista de comportamentos individuais.
  - Recebe como parâmetro uma lista de comportamentos individuais.

- def activar(self, percepcao: Percepcao) -> Accao:
  - Este método é responsável por ativar o comportamento composto com base na percepção atual do agente.
  - Itera sobre cada comportamento na lista de comportamentos e ativa cada um deles, obtendo as ações resultantes.
  - As ações resultantes de cada comportamento individual são armazenadas numa lista.
  - Após a ativação de todos os comportamentos individuais, o método seleccionar_accao é chamado para selecionar a ação final a ser tomada com base nas ações individuais.
  - Se não houver ações na lista, nenhum resultado é retornado.

- @abstractmethod
- def seleccionar_accao(self, accoes: list[Accao]) -> Accao:
  - Este método é marcado como abstrato usando o @abstractmethod. Isso significa que qualquer classe que herde ComportComp deve fornecer uma implementação para este método.
  - Recebe como parâmetro uma lista de ações individuais geradas pelos comportamentos.
  - Não há implementação neste método, apenas a declaração pass, indicando que ele deve ser implementado nas subclasses que herdam ComportComp.

Estimulo – Interface que define a funcionalidade geral de um estímulo (concretizado em função do problema a resolver)

- class Estimulo(ABC):
  - A classe Estimulo é definida como uma subclasse de ABC (Abstract Base Class), o que indica que é uma classe abstrata.

- @abstractmethod
- def detectar(self, percepcao: Percepcao) -> float:
  - Este método é marcado como abstrato usando o @abstractmethod. Isso significa que qualquer classe que herde Estimulo deve fornecer uma implementação para este método.
  - O método detectar recebe como parâmetro um objeto de percepção e deve retornar um valor float que representa a intensidade do estímulo detectado.
  - Não há implementação neste método, apenas a declaração pass, indicando que ele deve ser implementado nas subclasses que herdam Estimulo.

PACKAGE ECR:

Resposta – Implementação do mecanismo base de uma resposta

- class Resposta:
  - Esta classe representa a resposta gerada a partir de um estímulo, em termos da ação a ser realizada e da prioridade associada.

- def __init__ (self, accao: Accao):
  - O método __init__ é o construtor da classe. Ele inicializa uma instância de Resposta.
  - Recebe um parâmetro accao, que é a ação a ser associada à resposta.

- def activar(self, percepcao: Percepcao, intensidade: float = 0) -> Accao:
  - Este método ativa a resposta ao estímulo.
  - Recebe dois parâmetros: percepcao, que representa a percepção atual do ambiente, e intensidade, que é a intensidade do estímulo (com um valor padrão de 0.0).
  - Define a prioridade da ação com base na intensidade do estímulo.
  - Retorna a ação associada à resposta.

Hierarquia, que é uma subclasse de ComportComp. Os comportamentos estão organizados numa hierarquia fixa de subsunção (supressão e substituição)

- class Hierarquia(ComportComp):
  - Esta classe representa um tipo específico de comportamento composto, onde a seleção da ação é feita com base na prioridade das ações individuais.

- def seleccionar_accao(self, accoes):
  - Este método é responsável por selecionar a ação mais prioritária de uma lista de ações possíveis.
  - Recebe um parâmetro accoes, que é a lista de ações possíveis, ordenadas por prioridade.
  - Verifica se a lista de ações está vazia e, se estiver, retorna None.
  - Retorna a primeira ação da lista, que é a ação de maior prioridade.

Prioridade: Seleciona a ação de maior prioridade

- class Prioridade(ComportComp): Esta classe representa outro tipo de comportamento composto, onde a seleção da ação é feita com base na prioridade das ações individuais.

- def seleccionar_accao(self, accoes: list[Accao]) -> Accao:
  - Este método é responsável por selecionar a ação com a maior prioridade de uma lista de ações possíveis.
  - Recebe um parâmetro accoes, que é a lista de ações possíveis.
  - Verifica se a lista de ações está vazia e, se estiver, retorna None.
  - Utiliza a função max para encontrar o elemento com a maior prioridade na lista com base no atributo prioridade de cada objeto Accao.
  - Retorna o objeto Accao com a maior prioridade.

PACKAGE CONTROLO_REACT:

ControloReact: Herda da classe Controlo que fornece uma estrutura base para implementar o controle de um agente no ambiente simulado.

- class ControloReact(Controlo): É uma subclasse de Controlo, que é uma interface ou classe abstrata que define o comportamento esperado de um sistema de controle para um agente.

- def __init__(self, comportamento: Comportamento):

O construtor da classe recebe um objeto comportamento, que é um objeto de uma classe que herda de Comportamento. Este comportamento será ativado com base na percepção do ambiente.

- def processar(self, percepcao: Percepcao):

Este método recebe uma percepção do ambiente como entrada. Ele ativa o comportamento associado ao agente com base nesta percepção, chamando o método activar do comportamento. O resultado da ativação do comportamento é então retornado.

CONTROLO_REACT.REACCOES:

R**ecolher : Recolher, o qual é um comportamento composto que agrega um conjunto de sub-comportamentos (AproximarAlvo, EvitarObst, Explorar), os quais correspondem a sub-objectivos que é necessário concretizar para que o objectivo principal seja concretizado, ou seja, para recolher alvos é necessário aproximar alvo, evitar obstáculos e explorar (quando não é detectado qualquer alvo).**

- class Recolher(Hierarquia):
  - Esta classe representa um comportamento específico para recolher um alvo.
  - Ela herda da classe Hierarquia, que por sua vez é uma classe de comportamento composto.

- def __init__(self, comportamentos):
  - O construtor da classe inicializa os comportamentos necessários para recolher um alvo.
  - Cria uma lista de comportamentos, que inclui instâncias das classes AproximarAlvo, EvitarObst e Explorar.
  - Chama o construtor da classe pai (Hierarquia) usando super().__init__(comportamentos) para inicializar o comportamento composto.

CONTROLO_REACT.REACCOES.APROXIMAR:

AproximarAlvo: representa um comportamento para aproximar o agente de um alvo em uma direção prioritária. Ela herda da classe Prioridade, que é uma classe de comportamento composto com seleção baseada em prioridade. O construtor da classe inicializa os comportamentos de aproximação em cada uma das quatro direções possíveis (NORTE, SUL, ESTE, OESTE) e os passa para o construtor da classe pai (Prioridade) usando super().__init__(comportamentos). Isso garante que o comportamento composto seja inicializado corretamente com os comportamentos fornecidos.

- class AproximarAlvo(Prioridade):
  - Esta classe representa um comportamento para aproximar o agente de um alvo em uma direção prioritária.
  - Ela herda da classe Prioridade, que é uma classe de comportamento composto com seleção baseada em prioridade.
- def __init__(self, comportamentos):
  - O construtor da classe inicializa os comportamentos de aproximação em diferentes direções.
  - Cria uma lista de comportamentos, que inclui instâncias das classes AproximarDir para as direções NORTE, SUL, ESTE e OESTE.
  - Em seguida, chama o construtor da classe pai (Prioridade) usando super().__init__(comportamentos) para inicializar o comportamento composto.

AproximarDir: representa um comportamento de aproximação do agente numa direção específica. Aqui está uma descrição do que está acontecendo:

- class AproximarDir(Reaccao):
  - Esta classe é uma subclasse de Reaccao, o que implica que representa uma associação entre um estímulo (detecção de um alvo em uma direção específica) e uma resposta (movimento do agente na direção do alvo).
- def init(self, direccao: Direccao):
  - Recebe um parâmetro direccao que especifica a direção em que o agente deve se mover para se aproximar do alvo.
  - Chama o construtor da classe pai (Reaccao) usando super().__init__(EstimuloAlvo(direccao), RespostaMover(direccao)), onde EstimuloAlvo é responsável por detectar a presença de um alvo na direção especificada e RespostaMover é responsável por mover o agente na mesma direção. Isso inicializa a reação com o estímulo associado à detecção de um alvo nadireção especificada e a resposta associada a mover o agente nessa direção.

EstimuloAlvo, que é uma subclasse de Estimulo.

- class EstimuloAlvo(Estimulo):
  - Esta classe representa um estímulo que detecta um alvo em uma direção específica.
- def __init__(self, direccao: Direccao, gama: float = 0.9) -> None:
  - Ele inicializa a direção do alvo (direccao) e um fator de desconto (gama, padrão é 0.9) que será aplicado à intensidade do estímulo.
- def detectar(self, percepcao: Percepcao) -> float:
  - Recebe uma percepção do ambiente e verifica se há um alvo na direção especificada. Se um alvo for detectado, o estímulo gera uma intensidade com base na distância até o alvo, utilizando o fator de desconto gama para calcular essa intensidade. Se não houver um alvo na direção especificada, o estímulo retorna 0.

CONTROLO_REACT.REACCOES.EVITAR:

EstimuloObst, é uma subclasse de Estimulo.

- class EstimuloObst(Estimulo):
  - Esta classe representa um estímulo causado pela percepção de um obstáculo em uma direção específica.

- def __init__(self, direccao: Direccao, intensidade: float = 1) -> None:
  - O método __init__ é o construtor da classe. Ele inicializa a direção do obstáculo (direccao) e a intensidade do estímulo (intensidade, padrão é 1).

- def detectar(self, percepcao: Percepcao) -> float:
  - O método detectar recebe uma percepção do ambiente e verifica se há um obstáculo na direção especificada. Se um obstáculo for detectado na direção especificada, o estímulo gera uma intensidade equivalente à intensidade definida no construtor. Se não houver um obstáculo na direção especificada, o estímulo retorna 0.

EvitarDir, é uma subclasse de Reaccao.

- class EvitarDir(Reaccao):
  - Esta classe representa uma reação para evitar um obstáculo em uma direção específica.

- def __init__(self, direccao: Direccao, resposta: RespostaEvitar):
  - O método __init__ é o construtor da classe. Ele inicializa a direção do obstáculo (direccao) e a resposta de evitação associada.
  - Ao inicializar, cria uma instância de EstimuloObst para detectar a presença de um obstáculo na direção especificada.
  - Também recebe um objeto RespostaEvitar, que contém a lógica para lidar com a evitação do obstáculo.
  - A chamada super().__init__(EstimuloObst(direccao), resposta) inicializa a classe base Reaccao com o estímulo e a resposta fornecidos.

EvitarObst, é uma extensão da classe Hierarquia, representa um comportamento composto para desviar de obstáculos.

- class EvitarObst(Hierarquia):
  - Esta classe representa um comportamento composto para desviar de obstáculos.

- def __init__(self):
  - Inicializamos uma lista de comportamentos de desvio de obstáculo em todas as direções possíveis.
  - Cada comportamento de desvio de obstáculo é uma instância de EvitarDir, que recebe uma direção específica e uma instância de RespostaEvitar correspondente.
  - As direções são definidas como Direccao.NORTE, Direccao.SUL, Direccao.ESTE e Direccao.OESTE.
  - Em seguida, chamamos o construtor da classe pai Hierarquia usando super().__init__(comportamentos) para inicializar o comportamento composto.

CONTROLO_REACT.REACCOES.EVITAR (CONT.):

RespostaEvitar, que é uma resposta para a reação de evitar obstáculos. Aqui está uma descrição do que está acontecendo:

- class RespostaEvitar(RespostaMover):
  - Esta classe herda da classe RespostaMover, que representa uma resposta para mover em uma direção específica.

- def __init__(self, dir_inicial: Direccao = Direccao.ESTE):
  - No método __init__, inicializamos a classe pai RespostaMover com uma direção inicial, que por padrão é para leste (Direccao.ESTE).

- def activar(self, percepcao: Percepcao, intensidade: float) -> Accao:
  - Sobrescreve o método da classe pai para lidar com o caso de contato com obstáculos.
  - Se houver contato com obstáculos na direção atual, procuramos uma direção livre chamando o método privado __direccao_livre. Se encontrarmos uma direção livre, ativamos a resposta de mover-se para essa direção. Caso contrário, retornamos None.
  - Se não houver contato com obstáculos na direção atual, chamamos o método activar da classe pai RespostaMover para continuar com o movimento na direção atual.

- def __direccao_livre(self, percepcao: Percepcao) -> Direccao:
  - O método privado __direccao_livre retorna uma direção aleatória entre as direções disponíveis, ou seja, aquelas que não têm contato com obstáculos.

CONTROLO_REACT.REACCOES.EXPLORAR:

Explorar: esta classe simula o comportamento de um agente que escolhe uma direção aleatória para se mover durante a fase de exploração.

- class Explorar(Comportamento):

  - Esta classe representa um comportamento de exploração para o agente.
  - Ela herda da classe Comportamento, que é uma classe abstrata que define a interface para o comportamento do agente.

- def activar(self, percepcao: Percepcao) -> Accao:

  - No método activar, a direção de exploração é escolhida aleatoriamente a partir das direções disponíveis (norte, sul, leste e oeste). Em seguida, é criada uma instância da classe RespostaMover com a direção escolhida e o método activar dessa instância é chamado com a percepção atual do agente, retornando a ação resultante da exploração.

CONTROLO_REACT.REACCOES.RESPOSTA:

- class RespostaMover(Resposta):
  - Esta classe representa a geração de uma resposta para mover o agente em uma determinada direção.
  - Ela herda da classe Resposta, que fornece a estrutura básica para uma resposta genérica.

- def __init__(self, direccao: Direccao):
  - O construtor da classe recebe uma direção como argumento e inicializa uma instância da classe Accao com essa direção.

- def activar(self, percepcao: Percepcao, intensidade = 0.0) -> Accao:
  - No método activar, a ação associada é retornada. Isso significa que a ação de mover na direção especificada é devolvida como resposta à percepção atual do agente.

CONTROLO_REACT.REACCOES.EXPLORAR:

ControloReact, implementação de um sistema de controlo reativo.

- class ControloReact(Controlo):
  - Esta classe representa o controlo reativo do agente.
  - Ela herda da classe abstrata Controlo, que define a interface para o controlo do agente.

- def __init__(self, comportamento: Comportamento):
  - O construtor recebe um objeto de comportamento como argumento e o armazena como atributo _comportamento.

- def processar(self, percepcao: Percepcao):
  - O método processar recebe uma percepção como entrada e chama o método activar do comportamento armazenado, passando a percepção como argumento. Isso permite que o comportamento atual seja ativado com base na percepção atual do agente. O resultado da ativação do comportamento é retornado como ação a ser executada pelo agente.

PACKAGE AGENTE:

AgenteReact herda da classe Agente, representa um agente reativo.

- def __init__(self):
  - Cria-se um objeto do controlo reativo ControloReact, ao qual é passado um comportamento composto Recolher que por sua vez contém o comportamento Explorar. Ou seja, o agente primeiro tenta recolher algo, mas se não houver nada para recolher, ele explorará o ambiente.
  - Após inicializar o controlo reativo, chama-se o construtor da classe pai (Agente), passando o controlo reativo como argumento.

- if __name__ == "__main__":
  - No bloco if __name__ == "__main__":, cria-se uma instância do AgenteReact e executa-a num simulador com duração de 1 unidade de tempo.

A**genteSimul, representa um agente para um ambiente simulado.**

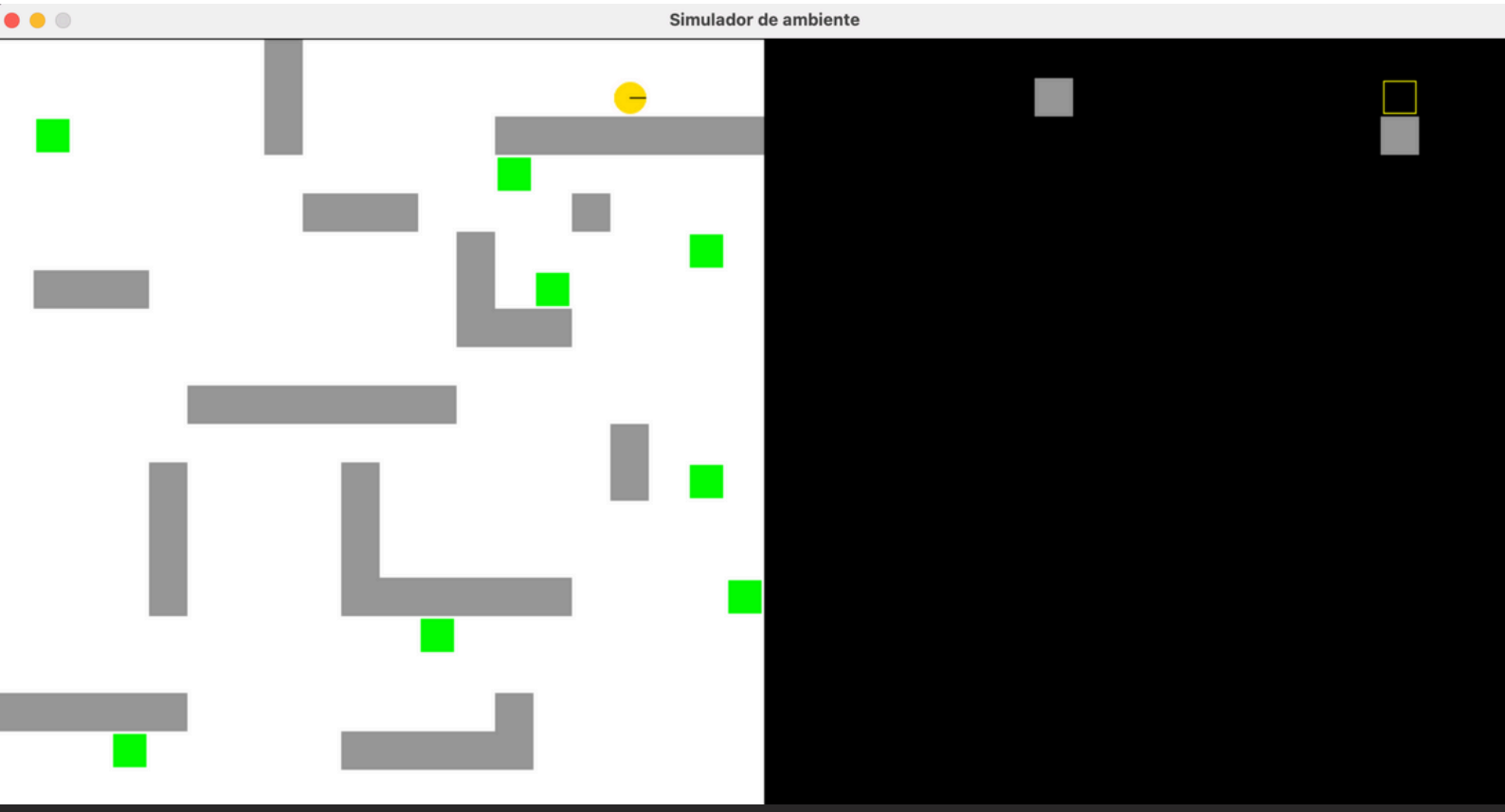
- def __init__(self):
  - No construtor, um objeto de controle ControloSimul é criado e passado para a classe pai (Agente).

- if __name__ == "__main__":
  - No bloco if __name__ == "__main__":, uma instância de AgenteSimul é criada e executada num simulador com duração de 1 unidade de tempo.

ControloSimul é uma subclasse de Controlo que incluí o método processar. Esta classe foi criada na primeira aula, para nos certificarmos que os imports e a pasta SAE estariam a funcionar corretamente.

33

PROJETO - PARTE 2 (AGENTE PROSPECTOR)



EXECUÇÃO DO PROGRAMA

PROJETO - PARTE 3 (AGENTE DELIB PEE)

OBJETIVO:

Agente Deliberativo PEE (Procura em Espaço de Estados)
Modelo de Planeamento:

- Baseia-se na procura num espaço de estados, onde o agente explora um grafo de estados a partir de um estado inicial até alcançar um estado objetivo.
- Utiliza algoritmos de procura como A*, Custo Uniforme, ou outros métodos heurísticos para encontrar o caminho ótimo.

Algoritmos Utilizados:

- Algoritmos de procura informada (como A*) ou não informada (como procura em largura ou profundidade).
- A procura é guiada por funções de custo e por heurísticas que estimam a distância até o objetivo.

Natureza da Solução:

- Gera uma sequência de ações específicas que levam do estado inicial ao estado objetivo.
- A solução é um plano detalhado de ações que deve ser seguido passo a passo.

Características do Ambiente:

- Adequado para ambientes onde o caminho até o objetivo pode ser claramente definido e seguido.
- A execução do plano assume que o ambiente é determinístico e que as ações têm resultados previsíveis.

PROCESSO:

Para começar o projecto – parte 3, percebi que pelo ficheiro “Projecto - Informação intercalar” disponibilizado no moodle, que o meu projeto continha erros tanto de sintaxe como de estrutura no meu trabalho, por isso revi todos os códigos e UMLs e consegui alterar alguns erros, assim como livrar-me de pastas extra que tinha criado para os imports do package SAE e etc... isto foi solucionado pela introdução na consola de: “export PYTHONPATH=/Users/tatianadamaya/Documents/ISEL/4ºSemestre/IASA/iasa50299/iasa_agente/src:/Users/tatianadamaya/Documents/ISEL/4ºSemestre/IASA/iasa50299/iasa_agente/src/lib”, antes de executar qualquer agente.
Para a execução do código, consultei os slides da ARQUITECTURA DE APOIO AO PROJECTO para garantir que todas as classes estariam bem organizadas, também consultei os slides, onde obtive algumas explicações sobre os packages que tinhamos de criar.

PACKAGE MOD

O package "mod" é composto por diversas classes que definem a estrutura básica para modelar problemas, operadores e estados num sistema de busca ou planeamento.

Classe "Problema" - classe abstrata que define a estrutura básica de um problema no sistema. As suas principais responsabilidades incluem armazenar o estado inicial do problema e a lista de operadores disponíveis.

- Métodos:
  - __init__(self, estado_inicial, operadores): Inicializa o problema com um estado inicial e uma lista de operadores.
  - estado_inicial: Getter para o estado inicial do problema.
  - operadores: Getter para a lista de operadores do problema.
  - objectivo(self, estado): Método abstrato para verificar se um estado atinge o objetivo do problema.

Classe "Operador" - classe abstrata que define a estrutura básica de um operador no sistema. Operadores são usados para transformar um estado noutro.

- Métodos:
  - aplicar(self, estado): Método abstrato que aplica o operador a um estado para gerar um novo estado.
  - custo(self, estado, estado_suc): Método abstrato que define o custo de transição de um estado para outro estado sucessor.

Classe "Estado" - classe abstrata que define a estrutura básica de um estado no sistema. Estados são elementos fundamentais que representam as diferentes configurações ou situações em que o sistema pode estar.

- Métodos:
  - id_valor(self): Método abstrato para obter o valor de identificação único do estado.
  - __hash__(self): Calcula o hash do estado com base no seu valor de identificação único.
  - __eq__(self, other): Verifica se dois estados são iguais com base no seu valor de identificação único.

Classe "EstadoAgente" - está num subpackage chamado agente de acordo com o UML, herda da classe "Estado" e representa o estado específico de um agente no ambiente, incluindo a sua posição atual.

- Métodos:
  - __init__(self, posicao): Inicializa o estado do agente com a posição fornecida.
  - posicao: Getter para a posição atual do agente.
  - id_valor: Retorna o valor de identificação único para este estado.

PACKAGE PEE

SUB-PACKAGE MEC_PROC

A classe MecanismoProcura é uma classe abstrata que define o esqueleto de um mecanismo genérico de procura. Ela define métodos para iniciar a memória, memorizar nós explorados e executar a procura no espaço de estados para encontrar uma solução. Ela expande os nós gerando sucessores utilizando operadores definidos no problema.
O método __init__ é o construtor da classe MecanismoProcura. Este inicializa o mecanismo de procura com uma fronteira, que é responsável por gerir os nós a serem explorados durante a procura.
O método procurar é o coração do mecanismo de procura. Ele recebe um problema como parâmetro e inicia a procura a partir do estado inicial do problema (problema.estado_inicial). Primeiro, ele inicializa a memória chamando _iniciar_memoria. Depois, cria um nó inicial (initial_node) e o insere-o na fronteira _fronteira.
Em seguida, entra num loop principal que continua enquanto a fronteira não estiver vazia (while not self._fronteira.vazia). A cada iteração, ele remove um nó da fronteira para explorar (current_node). Se o estado do current_node satisfizer o critério objetivo do problema (problema.objectivo(current_node.estado)), uma solução é encontrada e retornada (Solucão(current_node)).
Caso contrário, o método _expandir é chamado para gerar os nós sucessores do current_node usando os operadores definidos no problema. Cada nó sucessor gerado é então memorizado usando _memorizar(no_suc), o que pode envolver adicionar o nó aos nós explorados e à fronteira para continuar a procura.
O método _expandir é responsável por expandir um nó, gerando os seus nós sucessores usando os operadores definidos no problema. Para cada operador, ele aplica o operador ao estado atual do no (operador.aplicar(no.estado)) para obter um novo estado sucessor (estado_suc). Se o operador for aplicável ao estado atual, ele cria um novo nó sucessor (No(estado_suc, operador, no)) e retorna-o usando yield.

A classe Solucao representa um percurso correspondente a uma solução de um problema. Ela armazena a sequência de nós que compõem o percurso e permite acesso indexado e iteração sobre o percurso. Também é possível remover o primeiro nó do percurso.
O método __init__ inicializa um objeto Solucao a partir de um nó final (no_final). Ele constrói o percurso da solução retrocedendo a partir do nó final até o nó inicial (antecessor). Cada nó é inserido no início da lista __percurso, garantindo que o percurso seja mantido na ordem correta do início ao fim.
O método dimensão retorna a dimensão da solução, que é o número de nós no percurso (__percurso). Isso permite saber quantos nós compõem a solução encontrada para um problema.
O método remover remove e retorna o primeiro nó do percurso (__percurso). Esse método simboliza a continuação da exploração do percurso, removendo o próximo nó a ser explorado na procura.
Esses métodos especiais (__iter__ e __getitem__) permitem iterar sobre o percurso (__percurso) usando um loop for ou aceder elementos do percurso por índice, respectivamente. Eles facilitam a iteração e o acesso aos nós do percurso durante a solução de problemas.

A classe ProcuraGrafo estende MecanismoProcura e implementa um mecanismo de procura que explora um espaço de estados representado como um grafo. Ele mantém um dicionário de nós explorados e insere novos nós na fronteira se eles ainda não foram explorados.
O método _iniciar_memoria inicializa a memória necessária para a procura. Este chama o método _iniciar_memoria da classe pai (MecanismoProcura) e inicializa um dicionário _explorados para armazenar os nós que já foram explorados durante a procura.
O método _memorizar decide se um nó deve ser mantido na procura. Ele verifica se o estado associado ao nó ainda não foi explorado (_manter(no)). Se o estado não estiver nos nós explorados (_explorados), o nó é adicionado tanto aos nós explorados quanto à fronteira _fronteira para continuar a procura.
O método _manter verifica se o estado associado ao nó já foi explorado. Retorna True se o estado ainda não estiver nos nós explorados (_explorados), indicando que o nó deve ser mantido na busca.

A classe No representa um nó na árvore de procura. Cada nó mantém informações sobre o estado, o operador que gerou o estado, o nó antecessor, a profundidade do nó na árvore e o custo do percurso até o nó. A classe implementa a comparação entre nós com base no custo do percurso.
O método __init__ inicializa um objeto No com um estado, um operador (opcional) que gerou o estado, e um antecessor (opcional) que representa o nó antecessor na árvore de busca. Ele calcula a profundidade do nó na árvore de busca com base no antecessor e o custo do percurso até o nó.
O método __lt__ implementa a comparação de objetos No com base no seu custo. Isso permite que os nós sejam ordenados ou comparados com base no custo do percurso até o nó. Essa funcionalidade é essencial para algoritmos de busca que exigem a priorização de nós com custos menores.

PACKAGE PEE

SUB-PACKAGE MEC_PROC.FRONTEIRA

Cada classe de fronteira implementa uma estratégia específica para gerir a ordem de exploração dos nós durante a procura. A escolha da fronteira a ser usada afeta diretamente o comportamento e a eficiência da procura realizada pelo mecanismo de procura, permitindo adaptações para diferentes tipos de problemas e critérios de otimização.

A classe Fronteira serve como uma classe base abstrata para definir o comportamento básico das fronteiras de exploração no contexto de busca.
Inicializa a fronteira chamando o método iniciar, que é responsável por inicializar a estrutura de dados (_nos) que vai armazenar os nós a serem explorados.
Propriedade que verifica se a fronteira está vazia, retornando True se o comprimento da lista _nos for zero, indicando que não há mais nós para explorar.
Método para limpar a fronteira, reinicializando _nos como uma lista vazia. Isso é usado para preparar a fronteira para uma nova execução de busca.
Método abstrato que deve ser implementado por subclasses concretas. Define como um nó deve ser inserido na fronteira, mas deixa a implementação específica para as subclasses que definem diferentes estratégias de fronteira.
Remove e retorna o primeiro nó da fronteira, seguindo a ordem específica definida pela estratégia de inserção. No caso da classe base Fronteira, assume-se que o método inserir das subclasses determinará a ordem adequada para os nós.

Classe FronteiraFIFO (First-In, First-Out)
Esta fronteira mantém os nós numa fila, onde o primeiro nó inserido é o primeiro a ser removido.
Método inserir - Insere um nó na fronteira, adicionando-o ao final da lista _nos, mantendo a ordem de inserção. Isto garante que o primeiro nó inserido será o primeiro a ser removido, seguindo o princípio FIFO.

FronteiraLIFO (Last-In, First-Out)
Esta fronteira mantém os nós numa pilha, seguindo a ordem de inserção reversa. O último nó a ser inserido será o primeiro a ser retirado.

Classe FronteiraPrioridade
Esta fronteira mantém os nós numa fila de prioridade, onde os nós com menor prioridade são removidos primeiro.
Método __init__ - Inicializa a fronteira de prioridade com um avaliador, que é usado para determinar a prioridade dos nós na fila.
Método inserir - Insere um nó na fronteira com base na prioridade determinada pelo avaliador. Usa a função heappush do módulo heapq para inserir o nó na fila de prioridade, mantendo a ordem de prioridade.
Método remover - Remove e retorna o nó com a menor prioridade da fronteira de prioridade. Usa a função heappop do módulo heapq para remover o nó com a menor prioridade da fila de prioridade.

PACKAGE PEE

SUB-PACKAGE LARG

A classe ProcuraLargura herda de ProcuraGrafo, indicando que ela utiliza os métodos e atributos definidos na classe ProcuraGrafo, mas com algumas especificidades:

- No método __init__, o construtor da classe pai (ProcuraGrafo) é chamado utilizando super().__init__(FronteiraFIFO()).
- FronteiraFIFO() é passada como argumento para o construtor da classe pai, indicando que a fronteira usada nesta procura é baseada na estrutura FIFO.

Funcionamento da procura em Largura: procura em largura (Breadth-First Search, BFS) é uma técnica de procura em grafos que explora todos os nós de um determinado nível antes de passar para os nós do próximo nível. Usar uma fronteira FIFO garante que os nós sejam explorados nesta ordem, pois a fila insere os novos nós no final e remove os nós do início.

SUB-PACKAGE PROF

Estas classes são essenciais para implementar estratégias de procura em profundidade com diferentes abordagens de controlo de profundidade, sendo úteis em problemas onde é necessário explorar extensivamente um ramo da árvore de busca sem ficar preso em profundidades excessivas.

ProcuraProfundidade - implementa o algoritmo de procura em profundidade. Utiliza uma fronteira do tipo LIFO (Last-In, First-Out) para explorar os nós em profundidade até que não haja mais nós a serem explorados num determinado ramo da árvore de busca.
Métodos:
- __init__(self): Inicializa a busca em profundidade usando a fronteira LIFO.
- _memorizar(self, no): Memoriza um nó na fronteira LIFO para posterior exploração.

ProcuraProfLim - herda de ProcuraProfundidade e adiciona um limite máximo de profundidade (prof_max). Isso permite controlar até que profundidade o algoritmo de procura em profundidade deve ser executado.
Métodos:
- __init__(self, prof_max=100): Inicializa a busca em profundidade limitada com um limite máximo de profundidade. O padrão é 100.
- prof_max(self): Getter para o limite máximo de profundidade.
- prof_max(self, num): Setter para o limite máximo de profundidade.
- _expandir(self, problema, no): Expande os nós sucessores até o limite máximo de profundidade. Se a profundidade atual do nó for menor que `prof_max`, expande os nós sucessores normalmente.

ProcuraProfIter - implementa a procura em profundidade iterativa (Iterative Deepening Search). Realiza múltiplas procuras em profundidade com limites de profundidade crescentes até encontrar a solução.
Métodos:
- procurar(self, problema, inc_prof=1, limite_prof=100): Executa a busca em profundidade iterativa. Itera sobre limites de profundidade crescentes, realizando uma busca em profundidade com o limite atual. Retorna a solução encontrada ou None se não houver solução.

PACKAGE PEE

SUB-PACKAGE MELHOR_PRIM.AVAL

Esses avaliadores são essenciais para definir como os nós são priorizados durante a procura, permitindo adaptar os algoritmos de procura para diferentes tipos de problemas e estratégias, como procura de custo uniforme, A*, entre outros.

Avaliador (Classe Abstrata) - Define um avaliador para determinar a prioridade dos nós em algoritmos de busca.
Métodos:
- prioridade(self, no): Método abstrato para calcular a prioridade de um nó.
AvaliadorHeur - Implementa a definição de uma heurística para estimar o custo do caminho até o objetivo.
Métodos:
- definir_heuristica(self, heuristica): Método para definir a heurística a ser usada.
Args:
- heuristica - A heurística a ser definida para estimar o custo do caminho até o objetivo.

AvaliadorCustoUnif - Implementa o método de prioridade para a Procura de Custo Uniforme.
Métodos:
- prioridade(self, no): Calcula a prioridade de um nó para a Procura de Custo Uniforme.
AvaliadorAA . Implementa o método de prioridade para o algoritmo A*.
Métodos:
- prioridade(self, no): Calcula a prioridade de um nó para o algoritmo A*, que é a soma do custo acumulado até o nó com a estimativa do custo do nó até o objetivo (heurística).
AvaliadorSof - Implementa a heurística para o algoritmo de procura A*.
Métodos:
- prioridade(self, no)`: Calcula a prioridade de um nó para o algoritmo de procura A*, determinada pela heurística que estima o custo do nó até o objetivo.

SUB-PACKAGE MELHOR_PRIM

Essas classes encapsulam diferentes estratégias de procura informada, cada uma com suas peculiaridades em relação ao cálculo de prioridades dos nós e à utilização de heurísticas para guiar a busca em direção ao objetivo.

Heuristica (Classe Abstrata) - Define uma heurística para algoritmos de procura informada.
Métodos:
- h(self, estado): Método abstrato para calcular o valor heurístico de um estado.
ProcuraMelhorPrim - Representa a Procura de Melhor Primeiro, uma busca informada que utiliza uma fronteira de prioridade.
Métodos:
- __init__(self, avaliador): Inicializa a Procura de Melhor Primeiro com uma fronteira de prioridade.
- _manter(self, no`: Decide se um nó deve ser mantido na busca com base na sua prioridade e nos nós explorados anteriormente.
ProcuraInformada - Representa a Procura Informada, uma subclasse de ProcuraMelhorPrim usada para realizar procuras heurísticas.
Métodos:
- procurar(self, problema, heuristica): Inicia a busca informada, definindo a heurística a ser usada e chamando o método de procura da classe pai.
ProcuraAA - Representa o algoritmo A*, que utiliza uma função de avaliação combinando custo acumulado com heurística.
Métodos:
- __init__(self): Inicializa a Procura A* com um avaliador específico (AvaliadorAA).
ProcuraCustoUnif - Representa o algoritmo de Procura de Custo Uniforme, que expande sempre o nó com o menor custo acumulado até o momento.
Métodos:
- __init__(self): Inicializa a Procura de Custo Uniforme com um avaliador específico (AvaliadorCustoUnif).
ProcuraSofrega - Representa a Procura Sofrega, que utiliza uma heurística para estimar o custo do nó até o objetivo.
Métodos:
- __init__(self): Inicializa a Procura Sofrega com um avaliador sofrega (AvaliadorSof).

PROJETO - PARTE 4 (AGENTE DELIB PEE/ DELIB PDM)

OBJETIVO:

Agente Deliberativo PDM (Processo de Decisão de Markov)
Modelo de Planeamento:

- Baseia-se em Processos de Decisão de Markov (PDM), onde o agente considera as transições de estados e as recompensas associadas a cada ação num estado.
- Utiliza políticas de decisão que maximizam a utilidade esperada, considerando todas as possíveis transições e recompensas.

Algoritmos Utilizados:

- Algoritmos de iteração de valor ou iteração de política, que calculam a utilidade dos estados e determinam a melhor ação a ser tomada em cada estado.
- Utiliza um fator de desconto para avaliar a utilidade futura e convergir para uma política ótima.

Natureza da Solução:

- Gera uma política que define a melhor ação a ser tomada em cada estado para maximizar a utilidade total esperada.
- A política é um conjunto de regras que orienta o agente em qualquer estado em que ele possa se encontrar.

Características do Ambiente:

- Adequado para ambientes estocásticos onde as ações podem ter resultados incertos.
- Considera o longo prazo, avaliando recompensas futuras e adaptando-se a ambientes dinâmicos e incertos.

PROCESSO:

A implementação do Agente Deliberativo PDM (Processo de Decisão de Markov) revelou-se particularmente desafiante. Diferentemente de outros paradigmas de planeamento, o PDM exige uma compreensão profunda de conceitos matemáticos e probabilísticos subjacentes, o que tornou necessário um estudo extensivo antes de iniciar a codificação. Para compreender plenamente o processo de decisão de Markov, foi-me necessário revisar os slides várias vezes e realizar pesquisas detalhadas em fontes informativas diversas. Vídeos explicativos sobre PDM e algoritmos de iteração de valor e política também foram recursos valiosos que ,e auxiliaram na compreensão dos mecanismos. Somente após esse esforço preparatório intensivo foi possível começar a execução do código com segurança e precisão, garantindo que todos os aspectos teóricos fossem corretamente traduzidos numa implementação funcional, mesmo sabendo que nem todas as entregas foram perfeitas, mas concluindo com praticamente todos os objetivos esperados.

PACKAGE CONTROLO_DELIB

ControloDelib - Controlador Deliberativo, responsável por gerir o comportamento de um agente deliberativo. Utiliza processos de planeamento suportados por representações internas do ambiente.
Métodos:
- __init__(self, planeador): Inicializa o controlador deliberativo com o planeador responsável por gerar planos de ação.
- processar(self, percepcao): Processa a percepção do ambiente e decide a ação a ser tomada.
- __assimilar(self, percepcao): Atualiza o modelo do mundo com base na percepção.
- __reconsiderar(self): Decide se é necessário reconsiderar o plano atual.
- __deliberar(self): Gera um conjunto de objetivos com base na deliberação.
- __planear(self): Gera um novo plano de ação com base nos objetivos.
- __executar(self): Executa o plano de ação gerado.
- __mostrar(self): Mostra a representação visual do ambiente e do plano.

MecDelib - Mecanismo de Deliberação, responsável por gerar objetivos para o agente. Produz planos de ação com base nos objetivos gerados pelo mecanismo de deliberação.
Métodos:
- __init__(self, modelo_mundo): Inicializa o mecanismo de deliberação com a referência ao modelo do mundo.
- deliberar(self): Gera um conjunto de objetivos com base no modelo do mundo, retornando uma lista de estados que são considerados objetivos ordenados pela distância ao estado atual do agente.

SUB-PACKAGE MODELO

ModeloMundo - Representação interna do ambiente/problema. Utilizada no contexto de um agente deliberativo que toma decisões com base em processos de planeamento e raciocínio automático.
Métodos:
- __init__(self): Inicializa o modelo do mundo com atributos padrão, como estado atual, lista de estados conhecidos e operadores de movimento.
- alterado (property): Acede ao estado alterado.
- elementos (property): Acede aos elementos do ambiente.
- obter_estado(self): Obtém o estado atual do agente.
- obter_estados(self): Obtém todos os estados conhecidos.
- obter_operadores(self): Obtém os operadores disponíveis.
- obter_elemento(self, estado): Obtém o elemento na posição do estado especificado.
- distancia(self, estado): Calcula a distância entre o estado atual do agente e outro estado.
- actualizar(self, percepcao): Atualiza o modelo do mundo com base na percepção do ambiente.
- mostrar(self, vista): Mostra a representação visual do ambiente.

OperadorMover - Define a transformação (transição de estado) e o custo associado. Move o agente numa direção específica dentro do modelo do mundo, atualizando o estado conforme necessário e calculando o custo da transição.
Métodos:
- __init__(self, modelo_mundo, direcao): Inicializa o operador de movimento com a direção específica.
- accao (property): Acessa a ação do operador.
- ang (property): Acessa o ângulo da direção.
- aplicar(self, estado): Aplica a transformação de estado ao mover o agente.
- custo(self, estado, estado_suc): Calcula o custo da transição entre estados.

PACKAGE PLAN

Plano - Interface para os planos de ação.
Métodos:
- obter_accao(self, estado): Obtém a próxima ação a ser executada no estado especificado.
- mostrar(self, vista): Mostra uma representação do plano numa determinada visualização.

Planeador - Interface para os planeadores de ações.
Métodos:
- planear(self, modelo_plan, objetivos): Realiza o planeamento para alcançar os objetivos no modelo de plano especificado.

SUB-PACKAGE PLAN_PEE

PlanoPEE - Representa um plano de ação para o paradigma de Procura em Espaço de Estados (PEE).
Métodos:
- __init__(self, solucao): Inicializa o plano com uma solução obtida após a procura.
- obter_accao(self, estado): Obtém a próxima ação a ser executada no estado especificado.
- mostrar(self, vista): Mostra o plano na interface de visualização.

PlaneadorPEE - Planeador para o paradigma de Procura em Espaço de Estados (PEE).
Métodos:
- __init__(self, procura): Inicializa o planeador com um tipo específico de procura (A*, Custo Uniforme ou Gulosa).
- planear(self, modelo_plan, objectivos): Planeia a solução com base no modelo de planeamento e nos objetivos, retornando um plano de ação resultante da procura.

SUB-PACKAGE PLAN_PEE.MOD_PROB

ProblemaPlan - Dentro de outro subpackage mod_prob. Implementação de um problema específico para planeamento.
Métodos:
- __init__(self, modelo_plan, estado_final): Inicializa o problema com o modelo de planeamento e o estado final desejado.
- objectivo(self, estado): Verifica se o estado atual corresponde ao estado final desejado.

HeurDist Dentro de outro subpackage mod_prob. Implementação da heurística de distância euclidiana.
Métodos:
- __init__(self, estado_final): Inicializa a heurística com o estado final.
- h(self, estado): Calcula a distância euclidiana entre o estado atual e o estado final.

PACKAGE PLAN

SUB-PACKAGE PLAN_PDM

PlanoPDM - Representa um plano de ação gerado pelo Processo de Decisão de Markov (PDM).
Métodos:
- __init__(self, utilidade, politica): Inicializa o PlanoPDM com a utilidade e a política de ação
- obter_accao(self, estado): Obtém a ação associada a um estado na política de ação
- mostrar(self, vista): Mostra o plano de ação numa visualização

PlaneadorPDM - Classe responsável por planear usando PDM (Processo de Decisão de Markov)
Métodos:
- __init__(self, gama=0.85, delta_max=1.0): Inicializa o PlaneadorPDM com os parâmetros gama e delta_max
- planear(self, modelo_plan, objectivos): Planeia usando PDM e retorna o plano de ação resultante

SUB-PACKAGE PLAN_PDM.MODELO

ModeloPDMPlan - Combinação de um ModeloPDM e um ModeloPlan
Métodos:
- __init__(self, modelo_plan, objectivos, rmax=1000.0): Inicializa o ModeloPDMPlan
- obter_estado(self): Retorna o estado inicial do problema
- obter_estados(self): Retorna o conjunto de estados válidos
- obter_operadores(self): Retorna o conjunto de operadores (ações possíveis em cada estado) que definem transições de um estado para o outro e os seus custos
- S(self): Retorna o conjunto de estados válidos (método da interface ModeloPDM)
- A(self, s): Retorna o conjunto de operadores válidos (método da interface ModeloPDM)
- T(self, s, a, sn): Retorna a probabilidade de transição de um estado para outro dado um operador
- R(self, s, a, sn): Retorna a recompensa associada à transição de um estado para outro dado um operador
- suc(self, s, a): Retorna os sucessores de um estado dado um operador

SUB-PACKAGE MODELO

ModeloPlan - Interface para um modelo de planeamento
Métodos:
- obter_estado(self): Retorna o estado inicial do problema
- obter_estados(self): Retorna o conjunto de estados válidos
- obter_operadores(self): Retorna o conjunto de operadores (ações possíveis em cada estado) que definem transições de um estado para o outro e os seus custos

PACKAGE PLAN

SUB-PACKAGE PLAN_PDM

PlanoPDM - Representa um plano de ação gerado pelo Processo de Decisão de Markov (PDM).
Métodos:
- __init__(self, utilidade, politica): Inicializa o PlanoPDM com a utilidade e a política de ação
- obter_accao(self, estado): Obtém a ação associada a um estado na política de ação
- mostrar(self, vista): Mostra o plano de ação numa visualização

PlaneadorPDM - Classe responsável por planear usando PDM (Processo de Decisão de Markov)
Métodos:
- __init__(self, gama=0.85, delta_max=1.0): Inicializa o PlaneadorPDM com os parâmetros gama e delta_max
- planear(self, modelo_plan, objectivos): Planeia usando PDM e retorna o plano de ação resultante

SUB-PACKAGE PLAN_PDM.MODELO

ModeloPDMPlan - Combinação de um ModeloPDM e um ModeloPlan
Métodos:
- __init__(self, modelo_plan, objectivos, rmax=1000.0): Inicializa o ModeloPDMPlan, aqui tal como em vários métodos, certifiquei-me que os valores fossem double/float quando mencionados no UML.
- obter_estado(self): Retorna o estado inicial do problema
- obter_estados(self): Retorna o conjunto de estados válidos
- obter_operadores(self): Retorna o conjunto de operadores (ações possíveis em cada estado) que definem transições de um estado para o outro e os seus custos
- S(self): Retorna o conjunto de estados válidos (método da interface ModeloPDM)
- A(self, s): Retorna o conjunto de operadores válidos (método da interface ModeloPDM)
- T(self, s, a, sn): Retorna a probabilidade de transição de um estado para outro dado um operador
- R(self, s, a, sn): Retorna a recompensa associada à transição de um estado para outro dado um operador
- suc(self, s, a): Retorna os sucessores de um estado dado um operador

SUB-PACKAGE MODELO

ModeloPlan - Interface para um modelo de planeamento
Métodos:
- obter_estado(self): Retorna o estado inicial do problema
- obter_estados(self): Retorna o conjunto de estados válidos
- obter_operadores(self): Retorna o conjunto de operadores (ações possíveis em cada estado) que definem transições de um estado para o outro e os seus custos

PACKAGE PDM

MecUtil - Implementa o mecanismo de cálculo de utilidade para um PDM
Métodos:

- init(self, modelo, gama, delta_max): Inicializa o MecUtil com o modelo, fator de desconto (gama) e valor máximo de delta permitido
- utilidade(self): Calcula a utilidade para cada estado do modelo
- util_accao(self, s, a, U): Calcula o valor esperado de uma ação num estado dado

PDM - Implementa o processo de decisão de Markov
Atributos:

- __modelo: O modelo de processo de decisão de Markov
- gama: O fator de desconto
- delta_max: O limite de convergência
- __mec_util: O mecanismo de cálculo de utilidade

Métodos:

- init(self, modelo, gama, delta_max): Inicializa o PDM com o modelo, fator de desconto (gama) e valor máximo de delta permitido
- politica(self, U): Calcula a política de ação para cada estado possível
- resolver(self): Resolve o problema de decisão de Markov parcialmente observável

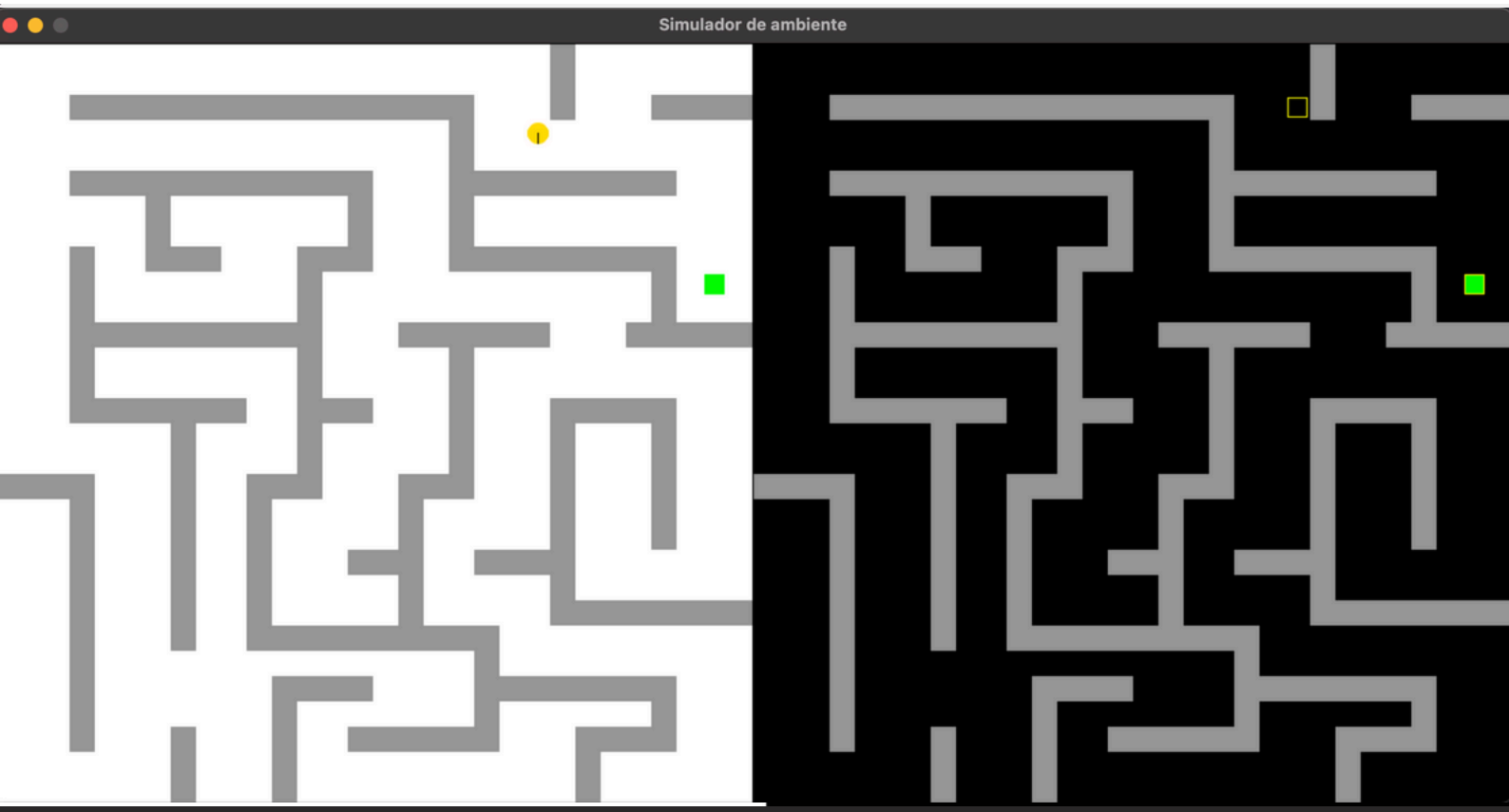
SUB-PACKAGE MODELO

ModeloPDM - Classe abstrata que define a estrutura básica de um modelo de processo de decisão de Markov (PDM)
Métodos Abstratos:

- S: Retorna o conjunto de estados do mundo
- A: Retorna o conjunto de ações possíveis num determinado estado
- T: Retorna a probabilidade de transição de um estado para outro através de uma ação
- R: Retorna o retorno esperado ao realizar uma transição de um estado para outro através de uma ação
- suc: Retorna o próximo estado dado um estado e uma ação

46

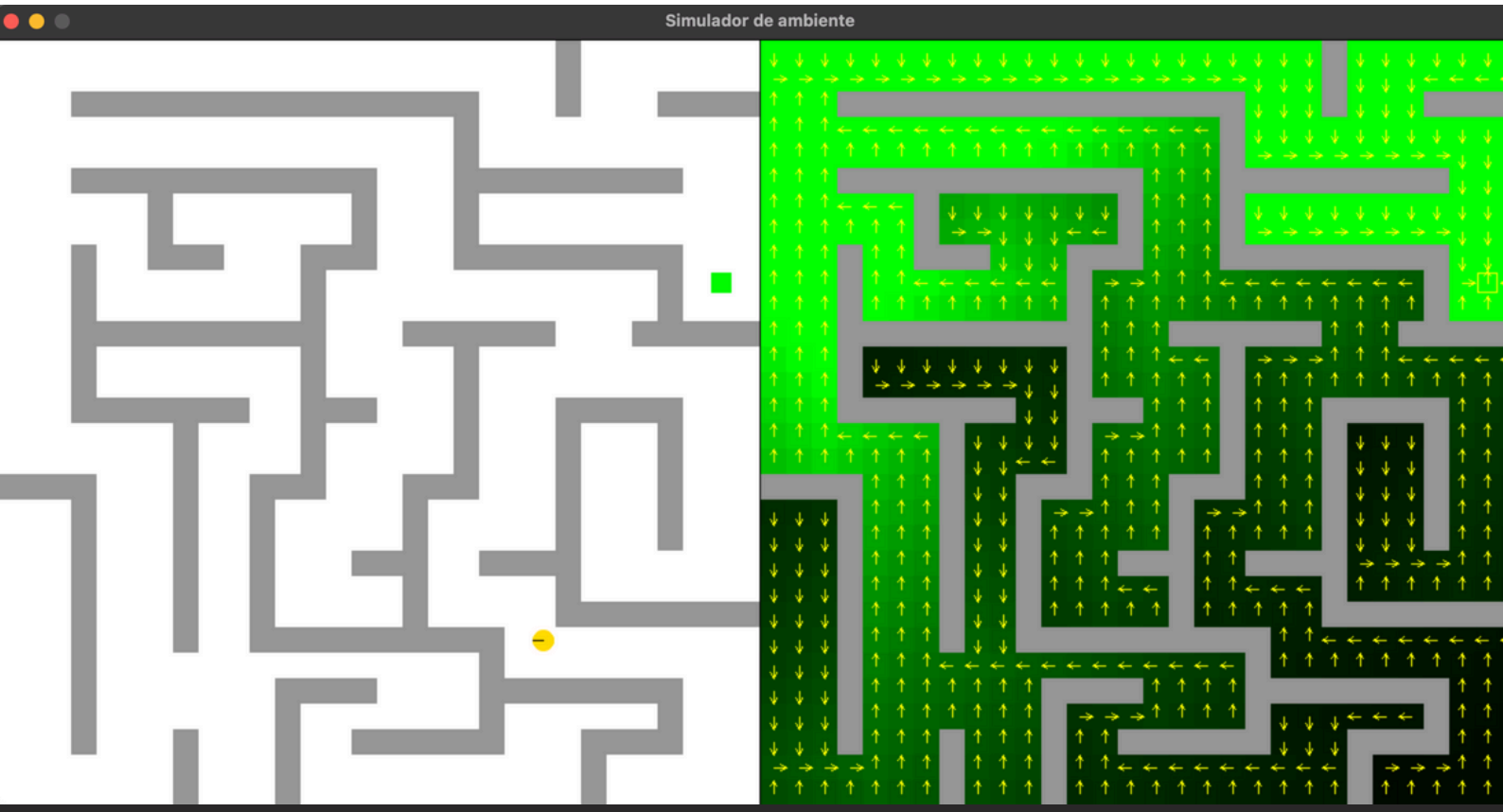
PROJETO - PARTE 4 (AGENTE DELIB PEE)



EXECUÇÃO DO PROGRAMA

47

PROJETO - PARTE 4 (AGENTE DELIB PDM)



EXECUÇÃO DO PROGRAMA

CONCLUSÃO

Para a conclusão deste relatório, escrevo que a execução deste trabalho trouxe-me uma melhor noção de como se processa a Inteligência Artificial e a importância da engenharia de software num programa, visto que foi a minha primeira abordagem na realização de programas através de praticamente somente linguagem UML.

Ler o enunciado e decifrar sozinha como poderia realizar o que estava a ser pedido da melhor forma, foi um processo demorado, mas que consegui atingir, através de várias pesquisas e tentativas até encontrar as soluções para os problemas, e uma vasta aprendizagem de novos métodos para executar determinadas ações.
Termino este trabalho/relatório também de uma forma realizada e mais apta e motivada para as próximas unidades curriculares em que estas abordagens sejam necessárias.