Relatório de Projeto

Inteligência Artificial para Sistemas Autónomos

Licenciatura em Engenharia Informática e de Computadores

Autores:	**Francisco Engenheiro**
ISEL ID:	49428
Professor:	**Phd Luís Morgado**
Ano académico:	2023/2024

Índice

Capítulo 1

Introdução

Este relatório documenta o projeto desenvolvido no âmbito da unidade curricular de Inteligência Artificial para Sistemas Autónomos (IASA), da Licenciatura em Engenharia Informática e de Computadores (LEIC).

O projeto tem como objetivo a aprendizagem de conceitos de inteligência artificial e a sua aplicação no desenvolvimento de sistemas autónomos.

1.1 Organização do Documento

O presente documento está organizado da seguinte forma:

- **Enquadramento Teórico**: Descrição dos conceitos de inteligência artificial e de sistemas autónomos, que servem de base ao projeto realizado;
- **Projeto**: Descrição do projeto desenvolvido, dividido em quatro partes, cada uma com um objetivo específico:
  - **Parte 1**: Desenvolvimento de uma biblioteca em *java* que fornece os mecanismos base para a implementação dos subsistemas que representam os conceitos gerais de inteligência artificial (e.g., agente, ambiente) e outros conceitos relacionados (e.g., máquina de estados); Desenvolvimento de um jogo que integra essa biblioteca (ver capítulo 3);
  - **Parte 2**: Desenvolvimento de um agente reativo em *python* com módulos comportamentais para recolher alvos e evitar obstáculos num ambiente com dimensões fixas (ver capítulo 4).
  - **Parte 3**: Desenvolvimento de uma biblioteca para procuras em espaços de estados (PEE) em *python* que envolveu a implementação de diferentes estratégias de procura; Desenvolvimento de uma biblioteca para modelação de problemas de procura em *python* que permite a definição de problemas concretos de forma independente

da estratégia de procura a aplicar; Modelação de um problema concreto de procura (ver capítulo 5).

Cada parte do projeto está organizada de forma a descrever o objetivo principal no contexto geral do projeto; fazer uma síntese dos conceitos estudados que se acharam relevantes para a sua implementação; caracterização do ambiente onde o agente vai atuar; descrever a implementação realizada, nomeadamente, a arquitetura do agente usada; justificar as principais decisões tomadas e a sua relação com os conceitos estudados; e apresentar a estrutura do código desenvolvido.

Existe um capítulo dedicado à revisão do projeto realizado, onde são identificados e descritos os erros cometidos nas entregas realizadas, com a indicação do problema e da respetiva correção, justificada com base nos conhecimentos adquiridos.

O projeto foi realizado individualmente e as entregas foram realizadas semanalmente, de forma incremental, de acordo com o plano de trabalho definido.

Capítulo 2

Enquadramento Téorico

A inteligência artificial, um ramo da engenharia informática, concentra-se no desenvolvimento de algoritmos e sistemas que imitam a capacidade humana de raciocínio. Entre as suas diversas aplicações, destacam-se a visão computacional, o processamento de linguagem natural, a robótica e a aprendizagem automática (*machine learning*). No contexto da aprendizagem automática, mais concretamente para o desenvolvimento de sistemas autónomos, ao qual este projeto se insere, destacam-se os conceitos de inteligência e autonomia. A inteligência caracteriza-se pela relação entre a cognição (i.e. capacidade de realizar a ação adequada dadas as condições do ambiente) e a racionalidade (i.e., capacidade de decidir no sentido de conseguir o melhor resultado possível perante os objetivos que se pretende atingir dado o conhecimento disponível). Já a autonomia é a habilidade de um sistema operar de forma independente, sem intervenção externa, seja ela humana ou de outro sistema. Representa uma característica da inteligência e portanto ser autónomo não significa ser inteligente, no entanto, ser inteligente implica ser autónomo [**?**].

2.1 Agente Inteligente

O modelo de um agente inteligente é uma abstração que permite representar a interação de um sistema com o ambiente onde opera. Este implementa um ciclo realimentado percepção-processamento-ação (ver figura 2.1), através do qual é realizado o controlo da função do sistema de modo a concretizar a finalidade desse sistema (e.g., a travagem automática de um automóvel quando deteta um obstáculo à sua frente) [**?**].

Além da autonomia, as caracteristicas principais de um agente inteligente, que estão dependentes do tipo de arquitetura (ver secção 2.4), são:

- **Reactividade**: Capacidade de reagir aos diversos estímulos do ambiente;
- **Pró-actividade**: Capacidade de tomar a iniciativa em função dos seus objetivos;

- **Sociabilidade**: Capacidade de interagir com outros agentes de forma a atingir os seus objetivos, sejam eles individuais ou coletivos;
- **Finalidade**: Propósito que o agente deve atingir e ao qual todas as suas características contribuem para a sua concretização.

Um agente pode percecionar eventos do ambiente através de sensores e executador comandos no ambiente através de atuadores.

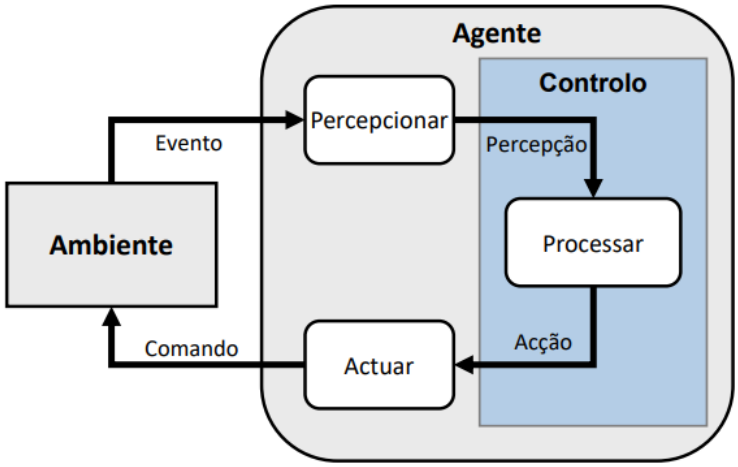


Figura 2.1: Representação conceptual da relação entre agente e ambiente. Retirado de [**?**], slide 30.

2.2 Ambiente

O espaço onde um agente opera é designado por ambiente, e que pode ser real, com sensores e atuadores físicos, (e.g., uma sala de aula) ou virtual (e.g., um jogo de computador). É caracterizado pelas seguintes propriedades [**?**]:

- **Totalmente Observável vs. Parcialmente Observável**: Um ambiente é totalmente observável se o agente tem acesso a uma descrição completa do ambiente e que lhe permite resolver o problema em questão sem necessitar de guardar estado interno; caso contrário, é parcialmente observável;
- **Tipo de Agente**: Os ambientes podem ser de agente único (i.e., apenas existe um agente a atuar) ou multi-agente (i.e., existem vários agentes a atuar, iguais ou diferentes entre si);
- **Determinístico vs. Estocástico**: Um ambiente é determinístico se o estado seguinte é unicamente determinado pelo estado atual e pela ação do agente; caso contrário, é estocástico. Mais ainda, dizemos que um ambiente continua a ser determinístico, mas mais concretamente estratégico, caso estejamos num ambiente multi-agente onde o

próximo estado pode estar também dependente das ações de outros agentes (e.g., num jogo de xadrez);

- **Episódico vs. Sequencial**: Um ambiente é episódico se a experiência do agente é dividida em episódios independentes; caso contrário, é sequencial (i.e., pode ser representado por uma sequência de estados);
- **Estático vs. Dinâmico**: Um ambiente é estático se não se altera enquanto o agente está a tomar uma decisão; caso contrário, é dinâmico;
- **Discreto vs. Contínuo**: Um ambiente é discreto se o número de estados possíveis é finito; caso contrário, é contínuo.

Na figura 2.2 são apresentados exemplos de caracterização de ambientes.

Ambiente	Observável?	Agentes?	Determinístico?	Episódico?	Estático?	Contínuo?
Palavras Cruzadas	Completamente Observável	Agente Único	Determinístico	Sequencial	Estático	Discreto
Poker	Parcialmente Observável	Multi-Agente	Estocástico	Sequencial	Estático	Discreto
Mundo Real	Parcialmente Observável	Multi-Agente	Estocástico	Sequencial	Dinâmico	Contínuo

Figura 2.2: Exemplos de caracterização de ambientes. Retirado de [**?**].

2.3 Agente Relacional

Um agente racional é um tipo de agente que realiza as ações corretas, ou seja, deve conseguir, a partir da exploração e aprendizagem, descobrir estratégias para chegar ao seu objetivo de forma ótima. Para esse efeito, é escolhida a ação que maximiza o valor esperado da medida de desempenho segundo a informação que lhe é fornecida (i.e., perceções) e o conhecimento adquirido até ao momento (e.g., conhecimento disponível sobre o ambiente).

O conceito de recolha de informação entra neste contexto, visto que a exploração pode ser feita fora do âmbito do objetivo com vista a adquirir conhecimento (e.g., estado do ambiente, ações possíveis, recompensas associadas a cada ação, etc) e, assim, melhorar a tomada de decisão seguinte.

No entanto, a racionalidade não implica clarividência, isto é, a ação tomada pode não resultar no que pretendemos ou esperamos, visto que o raciocínio nem sempre leva ao sucesso (e.g., o planeamento de uma viagem de avião, que foi reservada com antecedência e com atenção às condições climatéricas, não garante que o voo não seja cancelado no dia da viagem ou a viajem não seja interrompida por qualquer motivo alheio, mesmo tendo sido tomadas todas as precauções possíveis com base na informação disponível).

2.4 Arquiteturas de Agente

A arquitetura de um agente é a estrutura que define quais os componentes do agente (estrutura) e a forma como estes interagem entre si (comportamento). Um dos componentes de um agente é o módulo de controlo, que é responsável por processar perceções e gerar ações. A forma como este módulo é implementado determina o modelo de arquitetura do agente [**?**], que pode ser:

- **Reativo**: Associado a um paradigma comportamental, é caracterizado por associações diretas entre perceções e ações. A finalidade deste modelo é a concretização de objetivos implícitos (i.e., não estão explicitamente representados, por exemplo simbolicamente) presentes nas associações estímulo-resposta (i.e., as ações são diretamente ativadas em função das perceções);
- **Deliberativo**: Associado a um paradigma simbólico, é caracterizado pela existência de um módulo de deliberação que se situa entre a perceção e a ação. É neste módulo que ocorre o raciocínio e a tomada de decisão, com base em objetivos explícitos;
- **Híbrido**: Combinação dos modelos reativo e deliberativo, com o objetivo de tirar partido das vantagens que cada um oferece.

Capítulo 3

Projeto - Parte 1

Esta parte do projeto incidiu, principalmente, sobre desenvolvimento de uma biblioteca para providenciar abstrações dos subsistemas que representam conceito gerais de inteligência artificial (e.g., agente, ambiente) (ver figura 2.1) e outros conceitos relacionados (e.g., máquina de estados, conceito de transição de estado).

Para tal, foi necessário definir uma arquitetura de software que permitisse a implementação dos diferentes subsistemas de forma independente e modular, seguindo as diretrizes (i.e., métricas, princípios e padrões) que garantem a qualidade da arquitetura desenvolvida.

A implementação da biblioteca foi feita com base na consulta e compressão de diagramas UML e de sequência de forma a garantir a correta implementação dos diferentes subsistemas.

Por fim, foi desenvolvido um jogo que integra a biblioteca, permitindo a interação com o jogador por meio de comandos em texto.

3.1 Arquitetura de software

A arquitetura de software aborda a complexidade inerente ao desenvolvimento de software por meio de uma série de vertentes que estão interligadas.

3.1.1 Métricas

As métricas são medidas de quantificação da arquitetura de um software, indicadoras da qualidade dessa arquitetura [**?**].

O acomplamento é uma métrica inter-modular que mede o grau de interdependência entre os módulos de um sistema. Pode ser medido através da:

- **Direção**: Unidirecional vs Bidirecional (uni representa menos acoplamento);
- **Visibilidade**: Quando menor for a visibilidade de um módulo, menor é o seu acoplamento;

- **Relação**: (de mais acoplamento para menos) Herança → Composição → Agregação → Associação → Dependência (ver figura 3.1).

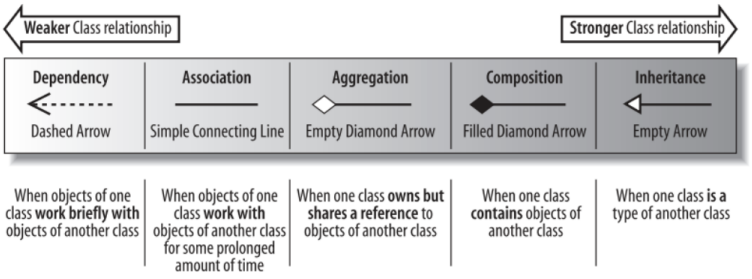


Figura 3.1: Relações entre classes e nível de acoplamento. Retirado de [**?**], slide 5.

A coesão é uma métrica intra-modular que determina o nível de coerência funcional de um subsistema/módulo, seja pela sua organização (i.e., cada modulo está organizado por conteúdo) ou pela sua funcionalidade (e.g., *single responsibility principle* - cada modulo tem uma única responsabilidade).

Destacam-se ainda outras métricas igualmente importantes, como a simplicidade e a adaptabilidade. A simplicidade representa, como o nome indica, a simplicidade da arquitetura desenvolvida, sendo que quanto mais simples for a arquitetura, mais fácil é a sua compreensão e manutenção. Já adaptabilidade mede a facilidade de alteração da arquitetura (i.e., adição de novos módulos ou alteração nos módulos existentes não deve implicar modificações significativas na arquitetura).

3.1.2 Princípios

Os princípios, no contexto da arquitetura de software, são um conjunto de convenções que orientam a sua definição no sentido de garantir a qualidade de produção da mesma, nomeadamente, no que se refere à minimização do acoplamento, à maximização da coesão e à gestão da complexidade [**?**]. Alguns exemplos são:

- **Abstração**: Define a forma como os componentes de um sistema são representados, permitindo a ocultação de detalhes de implementação;
- **Modularização**: Ao qual está associado a decomposição (e.g, divisão do sistema em sub-módulos) e o encapsulamento (i.e., ocultação de detalhes de implementação e/ou manutenção de estado privado e interno);
- **Factorização**: Onde a arquitetura é dividida em camadas, cada uma com um conjunto de responsabilidades bem definidas. Pode ser estrutural (e.g, Herança) e funcional (e.g, Delegação);

3.2 Processo de Desenvolvimento de Software

O processo de desenvolvimento de software consiste na criação da organização de um sistema de forma progressiva, através de diferentes níveis de abstração:

- **Modelo (Conceptual)**: Representação abstrata do sistema, que define o que o sistema deve fazer, sem especificar como;
- **Arquitetura (Modelo Concreto)**: Representação concreta do sistema, que define como o sistema deve ser implementado;
- **Implementação**: Código fonte que implementa o sistema definindo como o sistema deve ser executado.

Consiste num processo iterativo, em que as diferentes actividades de desenvolvimento são alternadas ao longo do tempo em função do conhecimento e do nível de detalhe envolvido [**?**]. Essa alternância poderá ser circular (i.e., implementação → arquitetura → modelo → implementação).

3.2.1 Tipos de Implementação

A implementação deve ser realizada em duas etapas principais: a implementação estrutural e a implementação comportamental (ver tabela 3.1).

Tabela 3.1: Tipos de Implementação

Tipo	**Modelo Associado**	**Designação**
Estrutural	UML	Define a estrutura de um sistema, ou seja, a forma como os componentes se relacionam entre si.
Comportamental	Diagrama de Sequência	Define o comportamento de um sistema, ou seja, a forma como os componentes interagem e comunicam entre si.

Mais detalhadamente, os diagramas de sequência ou atividade representam o fluxo de controlo de um sistema, ou seja, a sequência de atividades que um sistema executa e a sua ordem. Definem-se como modelos de interação com uma organização bidirecional (i.e., horizontal → tempo e vertical → estrutura) e são compostos por diferentes elementos de modelação (e.g., mensagens, operadores, linha de vida) [**?**].

Já a linguagem de modelação unificada (i.e., UML - *Unified Modeling Language*) representa um modelo de comportamento com interação como perspetiva principal de modelação. Este tipo de modelação descreve a forma como as partes de um sistema interagem entre si e com o exterior para produzir o comportamento do sistema. No contexto do projeto, esta linguagem também ajudou a compreender os conceitos fundamentais associados a um sistema

computacional (ver secção 3.3) através do suporte à representação de diagramas de transição de estado.

3.3 Modelação de um Sistema Computacional

Um sistema computacional geral, pode ser caracterizado de forma abstrata (ver figura 3.2), 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 [**?**].

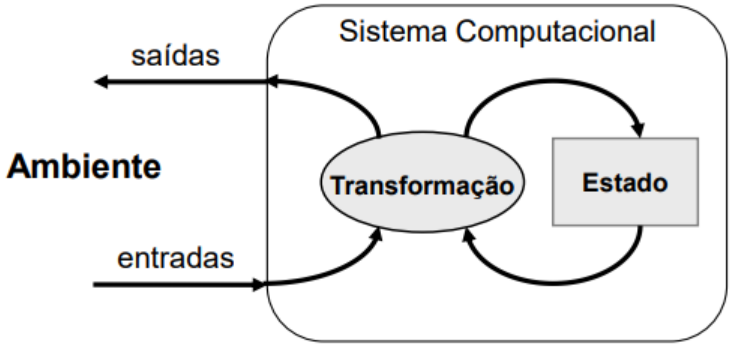


Figura 3.2: Modelo abstrato de um sistema computacional. Retirado de [**?**], slide 3.

Os conceitos fundamentais de um sistema computacional [**?**] são:

- **Dinâmica**: Descreve os estados que um sistema pode assumir e a forma como eles evoluem ao longo do tempo, determinando o comportamento do sistema;
- **Comportamento**: Expressa a estrutura de controlo do sistema, que corresponde à forma como o sistema age (gera as saídas) perante a informação proveniente do exterior (entradas) e do seu estado interno.

3.3.1 Modelo Formal de Computação

A função de transformação (i.e., responsável por gerar as saídas e o próximo estado em função das entradas e do estado actual do sistema), pode ser decomposta em duas funções distintas (ver tabela 3.2), em que **Q** representa o conjunto de estados possíveis do sistema (regularmente designado por espaço de estados), Σ representa o conjunto de entradas possíveis e **Z** representa o conjunto de saídas possíveis (ver figura 3.3).

Tabela 3.2: Funções de Transformação de um Sistema Computacional

Designação	**Expressão**	**Descrição**
Função de Transição (δ)	$\delta : Q \times \Sigma \rightarrow Q$	Define a relação entre o estado do sistema e as entradas que recebe, e o próximo estado.
Função de Saída (λ)	$\lambda : Q \times \Sigma \rightarrow Z$	Define a relação entre o estado do sistema e as entradas que recebe, e a saídas que produz.

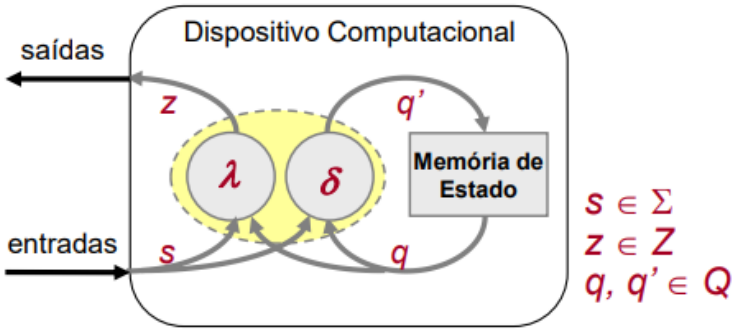


Figura 3.3: Modelo de um sistema computacional. Retirado de [**?**], slide 3.

Associado a um determinado modelo de um sistema computacional, está uma máquina de estados, que representa, de forma abstrata, o comportamento do sistema em função do tempo. Pode ser caracterizada por:

- **Estado**: Se o número de estados possíveis é finito, então a máquina de estados é finita; caso contrário, é infinita;
- **Determinismo**: Uma máquina de estados finita também pode ser subcaracterizada em determinística (i.e., existe apenas uma transição para o mesmo estado e entrada) ou não determinística (i.e., existe mais do que uma transição para o mesmo estado e entrada). Para qualquer máquina de estados não determinística, existe uma máquina de estados determinística equivalente; [**?** **?**]
- **Função de Saída**: Se a função de saída λ depende das entradas Σ, então a máquina de estados é do tipo *Mealy* ($\lambda : Q \times \Sigma \rightarrow Z$); caso contrário, é do tipo *Moore* ($\lambda : Q \rightarrow Z$).

3.4 Desenvolvimento do Jogo

Tendo por base os conceitos anteriormente abordados, que consolidaram a aprendizagem inerente à implementação da biblioteca, foi desenvolvido um jogo. No seu processo de desenvolvimento, foi definido um conjunto de componentes que providenciam implementações concretas dos subsistemas expostos pela biblioteca, adaptados ao contexto específico do jogo:

- **Personagem**: Representa o agente único do jogo;

- **Ambiente**: Representa o ambiente onde a Personagem opera;
- **Máquina de Estados**: Representa a máquina de estados associada ao controlo da Personagem. Caracterizada por ser finita, não determinística e do tipo *Mealy* (Ver figura 3.4).

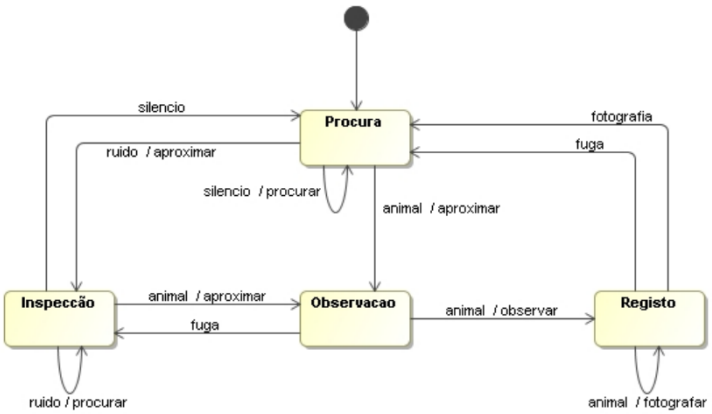


Figura 3.4: Máquina de estados associada ao controlo da Personagem. Retirado de [**?**], slide 14.

O jogo consiste num ambiente virtual onde a personagem tem por objectivo registar a presença de animais através de fotografias, simulando a exploração no mundo real. O ambiente pode ser caracterizado (ver secção 2.2) por ser:

- **Parcialmente observável**: A Personagem apenas sabe o que está no seu campo de visão, e por isso precisa de explorar o ambiente para descobrir o que está à sua volta, necessitando de guardar estado interno para tomar decisões (e.g., se observou um animal anteriormente, então pode registar a sua presença);
- **Agente único**: Considerou-se que a Personagem é a única a atuar no ambiente no contexto do problema, visto que os animais são tratados como objetos passivos e integrantes do ambiente, pois não apresentam objetivos próprios.
- **Estocástico**: Porque o estado seguinte pode depender de eventos aleatórios (e.g., a fuga de um animal) e não somente do estado atual e da ação da Personagem;
- **Sequencial**: Pois existe uma linha de acontecimentos que se sucedem (e.g., não é possível inspecionar, registar ou observar algo sem ser feita primeiro uma procura);
- **Dinâmico**: Pois o ambiente pode mudar enquanto a Personagem está a tomar uma decisão (e.g., os animais podem fugir);
- **Discreto**: Pois o número de estados possíveis é finito (e.g., pode ser mapeado para uma máquina de estados finita).

Foi criada uma aplicação (ver figura 3.5) de CLI (i.e., *command-line interface*) em *java*, que possibilita a interação com o jogador por meio de comandos em texto. Tendo em conta que a interface gráfica não era o foco desta parte do projeto, foi emulada através de texto descritivo.

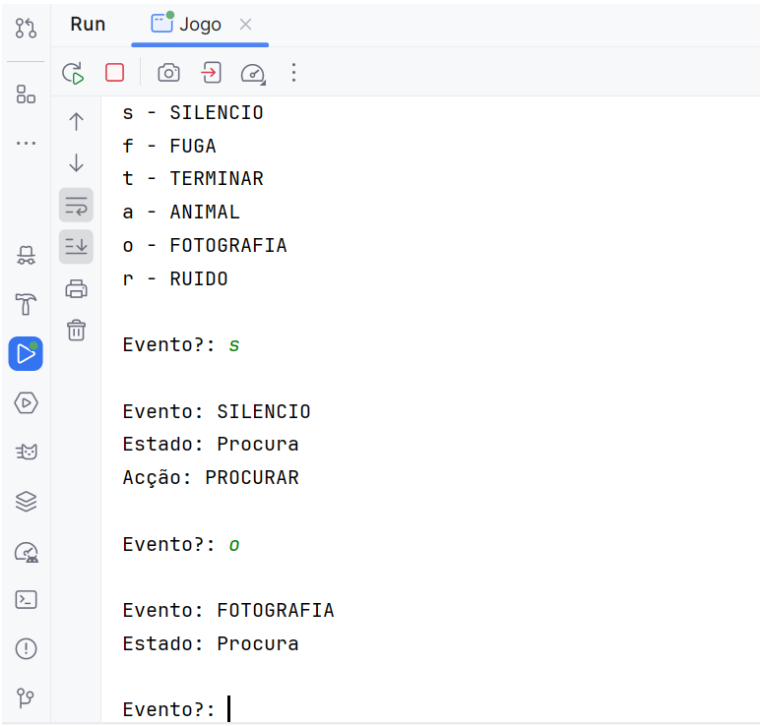


Figura 3.5: Utilização da aplicação do jogo.

3.5 Estrutura do Projeto

No processo de desenvolvimento de software associado a esta fase do projeto, foi definida a seguinte estrutura em módulos e que está presente na pasta *iasa_jogo/src*:

- *agente*: Integra classes e interfaces que definem o Agente e os seus componentes (e.g., módulo de controlo);
- *ambiente*: Agrega interfaces que representam o Ambiente e os seus componentes (e.g., comandos, eventos);
- *maqest*: Contém classes que definem o conceito de Máquina de Estados e os seus componentes (e.g., estados, transições);
- *jogo*: Agrega os detalhes da implementação do jogo, onde são integrados os módulos anteriores e definidas implementações concretas dos mesmos, adequadas ao contexto a que o jogo se insere.

Capítulo 4

Projeto - Parte 2

Nesta fase do projeto, o objetivo foi criar um sistema inteligente capaz de se movimentar num espaço com dimensões fixas onde existem obstáculos. A finalidade do agente prospetor é recolher os alvos e evitar os obstáculos presentes no ambiente (ver figura 4.1). O agente pode ser considerado homólogo a um robô móvel autónomo (e.g., um robô que aspira sozinho a casa), que se pode mover nos quatro sentidos cardeais (i.e., norte, sul, este e oeste).

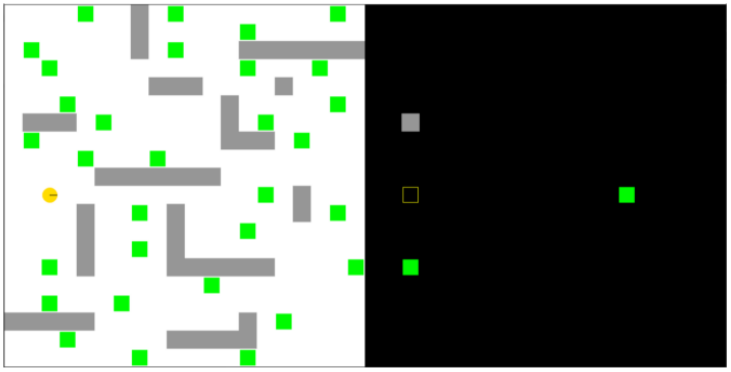


Figura 4.1: Representação visual desta fase do projeto. Retirado de [**?**], slide 13.

Para que o agente possa atingir os objetivos propostos, foi necessário implementar um agente reativo e os seus módulos comportamentais associados. O agente reativo foi implementado com base na biblioteca ECR (Esquemas Comportamentais Reativos) criada para esta fase do projeto, que expõe os mecanismos base de comportamento (e.g., reação, estímulo, resposta). A interação com o ambiente foi realizada através da biblioteca SAE (Simulador de Ambiente de Execução), que foi disponibilizada para esta fase do projeto.

Tal como em fases anteriores, a implementação foi feita com base na consulta e compressão de diagramas UML e de sequência de forma a garantir a correta implementação dos diferentes subsistemas.

4.1 Agentes Reativos

Na secção da arquitetura de agentes (ver secção 2.4), foi referido que uma arquitetura reativa é caracterizada pela associação direta entre perceções e ações. Associada a esta arquitetura está a ideia de que um agente reativo é composto por um conjunto de reações, e que são organizadas de forma modular em módulos comportamentais designados por comportamentos.

4.1.1 Reação

Inerente à arquitetura de agentes reativos, está o conceito de mecanismo de reação e que apresenta os seguintes elementos:

- **Estímulo**: Define a informação que é extraída de uma perceção, de forma a ser utilizada na ativação de uma resposta. Vários estímulos podem ser ativados por uma única perceção. Estão regularmente associados a um parâmetro de intensidade, que pode ser utilizado para determinar a prioridade de ativação de uma resposta;
- **Resposta**: Representa a geração de uma resposta a estímulos, inerentemente ligada a uma ação a ser executada e à sua respetiva prioridade. Além de poder ser ativada por um estímulo, uma resposta pode ser ativada diretamente por uma perceção, de forma a garantir restrições de ativação (i.e., guardas) se necessário;
- **Reação**: Módulo que associa estímulos a respostas.

Um agente reativo é composto por um conjunto de reações, que devem ser organizadas de forma modular em módulos comportamentais designados por comportamentos. Isto permite que as reações internas do agente sejam encapsuladas, facilitando a sua manutenção e escalabilidade.

4.1.2 Comportamento

Um comportamento (ver figura 4.2) define um módulo associado ao processamento de perceções do módulo de controlo de um agente reativo, onde é definida a forma como os objetivos implícitos do agente devem ser concretizados. Encapsula um conjunto de reações relacionadas entre si, possivelmente com uma sequência temporal, de forma a atingir um ou mais objetivos (e.g., evitar obstáculos, recolher alvos, etc). Associado a um objetivo podem existir sub-objetivos.

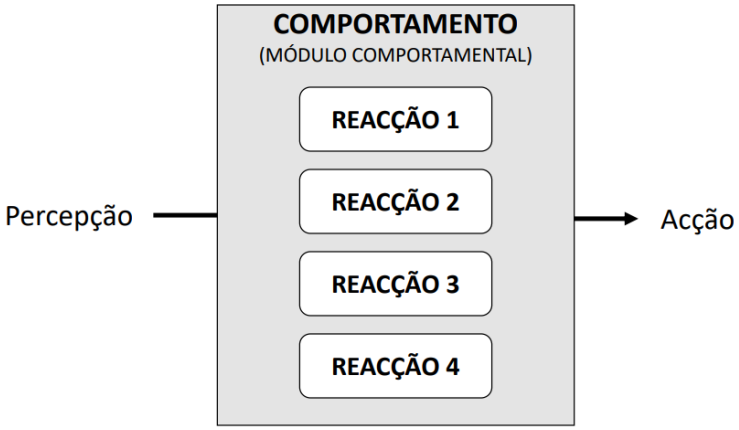


Figura 4.2: Módulo comportamental de um agente reativo. Retirado de [**?**], slide 11.

Tipos de Comportamento:

- **Comportamento Fixo**: Caracterizado por ter uma resposta fixa, pois gera uma ação em permanência;
- **Comportamento Simples**: Reação;
- **Comportamento Composto**: Agrega sub-comportamentos, ao qual está associado um mecanismo de seleção de ação de forma a determinar a ação a realizar em função das respostas dos mesmos.

4.1.3 Comportamento Composto

Como em comportamentos compostos, uma perceção pode ativar múltiplas reações, as quais geram diferentes ações, é necessário definir um mecanismo de seleção de ação. Alguns dos mecanismos de seleção de ação mais comuns são:

- **Execução paralela**: As ações são executadas em paralelo, porque não interferem entre si;
- **Seleção por prioridade**: As ações são selecionadas em função de uma prioridade. É possível distinguir dois tipos de comportamentos compostos com seleção por prioridade:
  - **Prioridade**: Representa uma forma de comportamento composto com um mecanismo de seleção de ação de prioridade dinâmica, e que portanto a prioridade dos comportamentos associados varia consoante o tempo (e.g., o comportamento "evitar obstáculo" tem prioridade sobre "explorar" quando o agente se encontra próximo de um obstáculo, mas "explorar" tem prioridade sobre "evitar obstáculo" quando não existem obstáculos próximos).
  - **Hierarquia**: Representa uma forma de comportamento composto com um me-

canismo de seleção de ação de prioridade por hierarquia fixa de subsunção (i.e., as camadas superiores controlam as inferiores, através da inibição, supressão e/ou reinício dos comportamentos associados). Os comportamentos mais altos na hierarquia têm prioridade sobre os mais baixos (correção de ação), e não variam consoante o tempo (e.g., o comportamento "carregar bateria", tem prioridade sobre os comportamentos "explorar" ou "evitar obstáculo", pois representa um comportamento mais básico e fundamental para a sobrevivência do agente, e que não varia consoante o tempo).

- **Combinação**: As ações são combinadas numa única resposta por composição (e.g., soma vectorial);
- **Seleção aleatória**: As ações são selecionadas aleatoriamente.

4.2 Controlo Reativo

Como foi apresentado na secção de arquiteturas de agentes (ver secção 2.4), um agente tem associado um módulo de controlo que é responsável por processar perceções e gerar ações. No caso de um agente reativo, o processar das perceções é realizado com base num módulo comportamental, também designado comportamento, o qual representa o comportamento geral do agente e que pode ser constituído por diferentes sub-comportamentos (ver figura 4.3) [**?**].

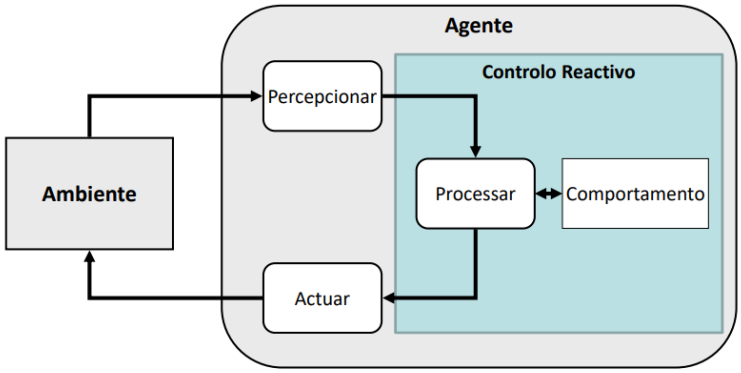


Figura 4.3: Agente com controlo reativo. Retirado de [**?**], slide 11.

4.3 Arquitetura Reativa com Memória

Numa arquitetura reativa com memória (ver figura 4.4), as reações dependem não só das perceções, mas também da memória de perceções anteriores (ou de informação delas derivada) para gerar as ações. Para esse efeito é necessário manter internamente memória, a qual é atualizada a partir das perceções e das reações ativadas, influenciando essas mesmas reações, bem como as ações geradas [**?**].

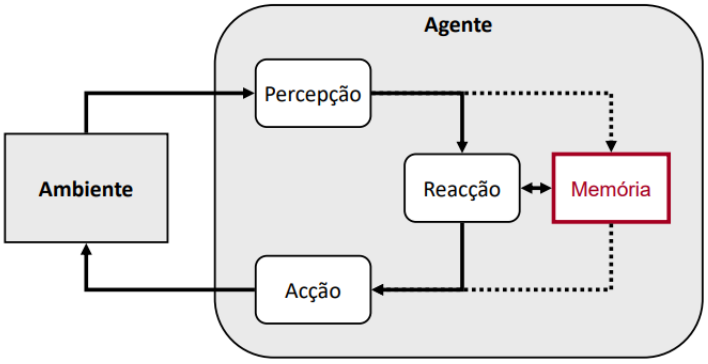


Figura 4.4: Arquitetura reativa com memória. Retirado de [**?**], slide 5.

O conceito de memória, associado a um comportamento, pode ser implementado de várias formas, como, por exemplo:

- **Estado**: As perceções anteriores, que sejam relevantes, são armazenadas num estado interno do comportamento (ver figura 4.5), que é atualizado e consultado em função das perceções atuais e que influencia a geracão de ações (e.g., o agente move-se numa direção aleatória até que a contagem dos passos efetuados seja superior a um determinado valor, e só depois é que muda de comportamento);
- **Máquina de Estados**: Além de ter em conta as perceções atuais e as reações correspondentes, o comportamento pode ter em consideração entradas ou até um mecanismo de reset, que influenciam a próxima transição de estado e a consequente geração de ações e possíveis saídas (ver figura 4.6). As entradas e saídas representam, neste contexto, interligações entre comportamentos [**?**].

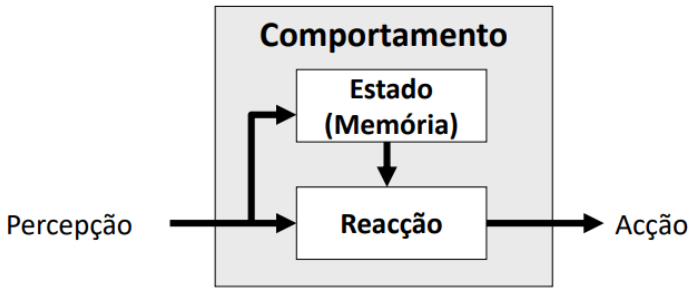


Figura 4.5: Comportamento com estado. Retirado de [**?**], slide 7.

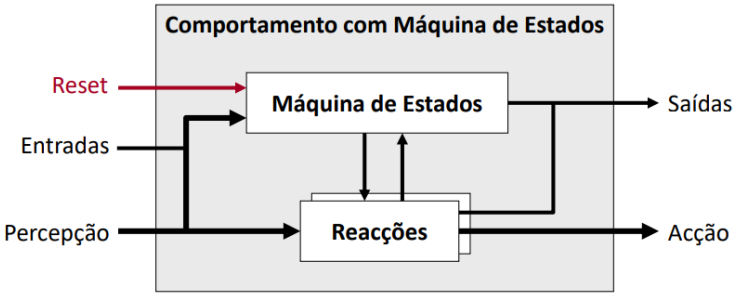


Figura 4.6: Comportamento com máquina de estados. Retirado de [?], slide 8.

Tabela 4.1: Vantagens e desvantagens da manutenção de estado em arquiteturas reativas. Retirado de [?].

Vantagens	**Desvantagens**
• Poder produzir todo o tipo de comportamentos; • Representar a evolução temporal do comportamento do agente; • Representar comportamentos complexos baseados na evolução temporal das perceções (e.g., agir devido à inexistência de um alvo após um determinado número de passos); • Lidar com falhas, explorando novas ações (e.g., escolher uma nova direção após embater num obstáculo).	• Aumento da complexidade espacial; • Aumento da complexidade computacional (i.e., manter as representações de estado); • Limitações no suporte de representações complexas e exploração de planos alternativos de ação.

4.4 Biblioteca SAE

Para esta fase do projeto foi disponibilizada uma biblioteca em *python* que apresenta e extende os conceitos expostos pela biblioteca realizada na primeira fase do projeto (ver capítulo 3). Além disso, a biblioteca SAE (Simulador de Ambiente de Execução) permite a execução de um agente autónomo num ambiente simulado bidimensional, composto por alvos e obstáculos [?]. Oferece, além de outras funcionalidades, a possibilidade de utilizar sete configurações de ambiente predefinidas.

A interface gráfica é composta por duas áreas de visualização (ver figura 4.7): a primeira área (à esquerda) apresenta a configuração do ambiente escolhida e com a indicação do agente; e a segunda área (à direita) é uma área de visualização de informação interna do agente.

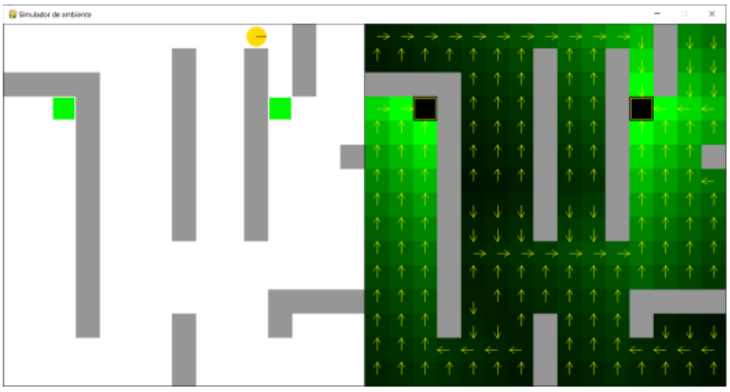


Figura 4.7: Interface gráfica do SAE. Retirado de [**?**], pág. 3.

4.5 Caracterização do Ambiente

O ambiente onde o agente reativo se movimenta é virtual, bidimensional, com dimensões fixas e composto por alvos e obstáculos estáticos. Pode ser caracterizado (ver secção 2.2) por ser:

- **Totalmente observável**: O agente tem acesso a toda a informação do ambiente;
- **De agente único**: O ambiente contém apenas um agente a atuar;
- **Determinístico**: O próximo estado é unicamente determinado pelo estado atual e pela ação do agente;
- **Sequencial**: As ações do agente afetam as etapas consequentes (e.g., a recolha de um alvo implica a sua remoção do ambiente);
- **Estático**: O ambiente não muda enquanto o agente está a decidir a próxima ação a realizar;
- **Discreto**: O ambiente é composto por um número finito de estados possíveis.

4.6 Implementação do Agente Reativo

No âmbito desta fase do projeto, foram realizadas as seguintes tarefas:

1. Definição de uma biblioteca denominada ECR (Esquemas Comportamentais Reativos) que expõe os mecanismos base de comportamento (ver secção 4.1.2), reação (ver secção 4.1.1), estímulo, resposta, e comportamento composto (ver secção 4.1.3);
2. Implementação de um conjunto de comportamentos e subcomportamentos que permitem ao agente autónomo atingir os objetivos propostos (i.e., recolher alvos e evitar obstáculos) utilizando a biblioteca ECR;

3. Observação do comportamento do agente e da sua interação com o ambiente através da interface gráfica da biblioteca SAE;

4. Adição de comportamentos mais complexos ao controlo do agente reativo de forma a melhorar o seu desempenho tendo em conta os objetivos propostos (e.g., inicialmente o agente apenas explorava numa direção aleatória, mas posteriormente foram adicionados comportamentos compostos de recolha de alvos e evitação de obstáculos); Ocorreu em simultâneo com o ponto **3**.

Em relação ao ponto **2**, foram implementados os seguintes comportamentos:

- **Recolher Alvo**: Representa um comportamento composto com um mecanismo de seleção de ação por hierarquia fixa de prioridade, que corresponde ao objetivo do agente prospector de recolher alvos. O agente deve, por esta ordem de prioridade, ter em consideração os seguintes subcomportamentos: (1) aproximar alvo; (2) evitar obstáculos; e (3) explorar o ambiente.

- **Aproximar Alvo**: Representa um comportamento composto com um mecanismo de seleção de ação por prioridade dinâmica, que corresponde ao objetivo do agente prospector de se aproximar de um alvo. Como a distância entre o agente e os diversos alvos varia consoante a posição do agente no ambiente (i.e., a intensidade do estímulo varia), o agente deve selecionar o alvo mais próximo (i.e., prioritizar o estímulo que apresenta maior intensidade) e por isso é que é usado um mecanismo de seleção por prioridade dinâmica (ver secção 4.1.3).

- **Evitar Obstáculo**: Representa um comportamento composto com um mecanismo de seleção de ação por hierarquia fixa de prioridade, que corresponde ao objetivo do agente prospector de evitar obstáculos. É usado este mecanismo de seleção de ação, porque a próxima direção livre de obstáculos é calculada em função da posição do agente e dos obstáculos presentes à sua volta, de forma aleatória. E portanto, num dado momento, não existe uma direção livre mais prioritária (no sentido literal) que outra, mas sim, uma hierarquia onde estão definidas, pela ordem de descoberta aleatória, as direções livres de obstáculos encontradas.

- **Explorar**: Define um comportamento fixo sem memória (e.g., representação interna de perceções anteriores) e que, por isso está condenado à repetição. Está associado ao objetivo "explorar" e caracteriza-se por ter uma resposta fixa, pois gera uma ação em permanência, que consiste em mover-se numa direção aleatória. Não depende de nenhum estímulo para ser ativado.

Além dos comportamentos descritos, para os comportamentos compostos, foram implementados estímulos, respostas e reações, de forma a garantir a descrição e execução correta dos

comportamentos associados.

Em paralelo ao desenvolvimento da implementação descrita e de forma a introduzir os conceitos de arquitetura reativa com memória, mais concretamente, o conceito de comportamento com estado (ver secção 4.3), foi introduzido um comportamento que não caracteriza nenhum objetivo inicial descrito. Este comportamento, denominado *Contar Passos*, caracterizado por ser um comportamento fixo com memória, foi implementado com a finalidade de contar os passos do agente e tomar decisões com base no número de passos efetuados (e.g., ao fim de um determinado número de passos, o agente muda de direção). No entanto, tal comportamento não consta na implementação final do agente reativo, pois não é relevante para os objetivos propostos.

4.7 Estrutura do Projeto

No processo de desenvolvimento de software associado a esta fase do projeto, foi definida a seguinte estrutura em módulos e que está presente na pasta *iasa_ agente/src*:

- *agente/agente_ reativo.py*: Integra a implementação do agente reativo;
- *agente/controlo_ react*: Integra a implementação do controlo do agente reativo, bem como os componentes dos módulos comportamentais associados;
- *lib/ecr*: Integra a implementação da biblioteca ECR;
- *lib/sae*: Integra a implementação da biblioteca SAE, que foi disponibilizada para esta fase do projeto;
- ficheiro *teste.py*: Executa o agente reativo implementado no ambiente simulado com a configuração escolhida, providenciado pela biblioteca SAE.

Capítulo 5

Projeto - Parte 3

Nesta fase do projeto, foram abordadas as temáticas de raciocínio automático e implementados mecanismos de procura em espaços de estados, que servem como base para a implementação de um futuro agente deliberativo. Foi também realizada uma biblioteca para modelação de problemas, servindo como argumento para que a aplicação de um determinado mecanismo de procura não fosse dependente da especificidade do problema.

No final desta fase, foi modelado um problema de procura concreto, de forma a aplicar as diferentes estratégias de procura implementadas e avaliar o seu desempenho.

Tal como em fases anteriores, a implementação foi feita com base na consulta e compressão de diagramas UML e de sequência de forma a garantir a correta implementação dos diferentes subsistemas.

5.1 Raciocínio Automático

O raciocínio automático é uma área da inteligência artificial que se dedica a desenvolver algoritmos capazes de inferir conclusões a partir de um conjunto de premissas. Pode ser visto como um processo computacional, onde se parte de uma representação de conhecimento de um determinado problema e se chega a conclusões, com base em regras de inferência (i.e., processo de manipulação de representações simbólicas para deduzir novas informações baseadas em conhecimento do domínio).

Associado ao processo de raciocínio automático, estão dois tipos de atividades principais:

- **Exploração**: A exploração das opções possíveis, e que requer antecipação e previsão das suas consequências (raciocínio prospetivo) (e.g., num determinado momento, quantas jogadas possíveis existem num jogo de xadrez para que o jogador possa ganhar o jogo);
- **Avaliação**: A avaliação das opções exploradas, que requer a comparação das opções possíveis e a escolha da melhor opção. Associadas a escolha estão medidas de desempe-

nho como o custo e o ganho (e.g., de todas as jogadas possíveis, qual a que maximiza a probabilidade de ganhar o jogo de xadrez).

De forma a resolver um determinado problema, através do raciocínio automático, é necessário representar o conhecimento do seu domínio através de representações simbólicas internas ao sistema - modelo do problema - com a informação necessária para a resolução do mesmo. Estas representações servem de base à simulação para exploração e avaliação de opções possíveis, e são obtidas através de processos de codificação (i.e., da informação concreta do problema em estruturas simbólicas internas) e de descodificação (i.e., processo inverso). Um exemplo de codificação e descodificação de um problema é ilustrado na figura 5.1.

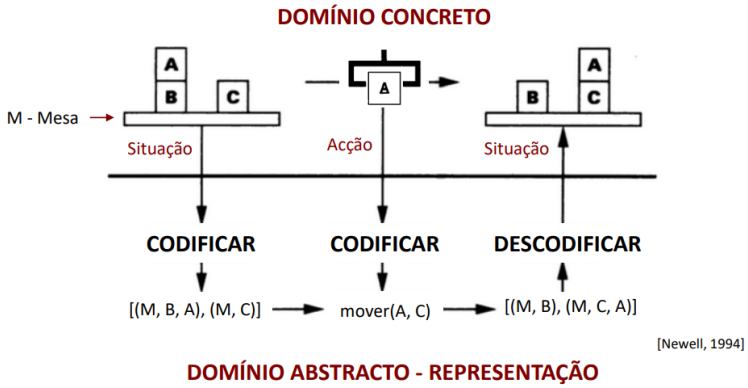


Figura 5.1: Representação de um problema. Retirado de [**?**], slide 8.

5.1.1 Modelação de um Problema

O modelo de um problema é uma representação simbólica interna ao sistema, que contém a informação necessária para a resolução do problema. O mecanismo de raciocínio automático é responsável por manipular esta representação, de forma a inferir conclusões e a resolver o problema. Esta representação tem como base os seguintes conceitos:

- **Estado**: Representação única e simbólica de um conjunto de informações concretas que caracterizam um estado da modelação de um problema. No âmbito do raciocínio automático, os estados abstraem os aspetos (configurações) estruturais que caracterizam o problema. Ao conjunto de estados possíveis de um problema e às transições entre eles, dá-se o nome de espaço de estados;
- **Operadores**: Representação de uma ação geradora de uma transição de um estado para outro. No âmbito do raciocínio automático, os operadores abstraem as transformações (dinâmicas) que ocorrem. Associado a um operador estão duas funções: uma que gera o estado sucessor a partir de um dado estado e outra que define o custo associado a essa transição, e que pode depender ou não do estado atual e do estado sucessor;
- **Transição**: Em contraste com o operador, a transição é a ação efetiva da passagem

de um estado para outro (geram o estado sucessor), que ocorre quando um operador é aplicado sobre as representações internas dos estados.

- **Problema**: Dá suporte ao raciocínio automático, modelando um problema ao qual está inerente uma finalidade; Associado a um problema está o estado inicial, um conjunto de operadores e uma função predicado [**?**] que verifica se um estado recebido é um estado objetivo (i.e., que satisfaz a finalidade do problema). O problema considera-se resolvido quando um estado objetivo é atingido.

5.2 Mecanismo de Procura

O mecanismo de procura é um dos mecanismos de raciocínio automático, que tem como objetivo encontrar uma solução para um problema de procura.

Associado a um mecanismo de procura está o conceito de árvore de procura, que é uma estrutura responsável por manter a informação gerada durante o processo de procura. Caracteriza-se pelos seguintes conceitos:

- **Nó**: Representa um estado do problema. Além disso, contém informação, tal como: operador (que foi aplicado para gerar o estado sucessor, e que pode não existir); antecessor (nó antecessor ao qual foi aplicado o operador para gerar este nó, e que pode não existir). E informação complementar para controlo do processo de procura: profundidade (do nó na árvore de procura, incremental a cada nível da árvore de procura); custo (associado ao caminho que vai do nó raiz ao nó, incremental a cada transição de um nó para o nó sucessor). Existe ainda o conceito de nós abertos (que ainda não foram expandidos) e nós fechados (que já foram expandidos); Os nós são comparáveis entre si através do seu custo;

- **Fronteira**: Representa os nós folhas da árvore de procura. Estes nós são candidatos a serem expandidos. A forma como é feita essa expansão dependerá da estratégia de procura utilizada (ver secção 5.3);

- **Solução**: Representa um percurso (sequência de estados e operadores), no espaço de estados, mais concretamente, corresponde ao caminho que vai do nó raiz (estado inicial) a um dos nós objetivo (estado objetivo) encontrado. Podem existir várias soluções para um problema de procura, que dependem também da estratégia de procura utilizada, ou até mesmo não existir solução.

5.2.1 Processo de Procura

O processo de procura (ver figura 5.2) é um processo iterativo que tem como objetivo encontrar uma solução para um problema de procura, através da exploração do espaço de estados a partir do estado inicial. É caracterizado, de forma abstrata, pelos seguintes passos [**?**]:

- É criado um nó para representar o estado inicial;
- O nó é memorizado (depende da estratégia de procura utilizada);
- Enquanto a fronteira não estiver vazia:
  - Remover um nó da fronteira;
  - Verificar se o estado que lhe está associado é um estado objetivo. Se for um estado objetivo, o processo de procura termina, e é devolvida a solução que contém esse nó;
  - O nó removido é expandido, gerando os nós sucessores;
  - Os nós sucessores são memorizados (depende da estratégia de procura utilizada).
- Se a fronteira estiver vazia, o processo de procura termina com a indicação de que não existe solução.

A expansão de um nó, mencionada no processo de procura anterior, é também um processo iterativo, que tem como objetivo gerar os nós sucessores de um nó. É caracterizada, de forma abstrata, pelos seguintes:

- Para cada operador, gerar a transição de estado do nó, que vai permitir obter o estado sucessor;
- Se o estado sucessor existir:
  - Calcular o custo do nó sucessor (custo do nó antecessor + custo da transição para o estado sucessor);
  - Criar o nó sucessor com o estado sucessor obtido, o operador que gerou a transição, o nó antecessor e o custo calculado anteriormente;
  - Guardar o nó sucessor gerado numa lista.
- Retornar a lista de nós sucessores gerados.

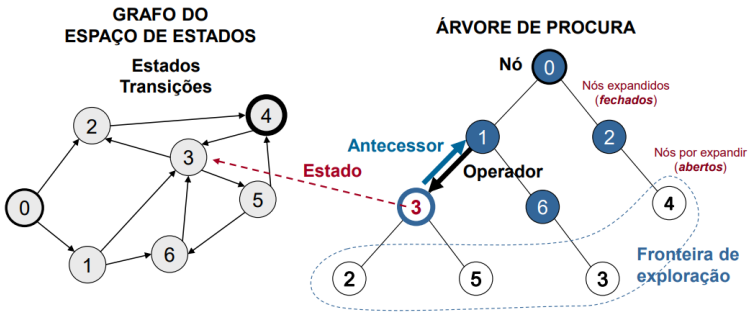


Figura 5.2: Processo de procura. Retirado de [**?**], slide 7.

5.3 Estratégias de Procura

As estratégias de procura são responsáveis por definir a forma como a árvore de procura é explorada, ou seja, a ordem pela qual os nós são expandidos. Existem várias estratégias de procura que se podem dividir em dois grandes grupos: estratégias de procura não informada (cega) (ver secção 5.3.2) e estratégias de procura informada (heurística) (ver secção 5.3.3). Além desta divisão, existe o conceito de procura em grafo (ver secção 5.3.1), que é uma técnica de procura que evita o aparecimento de ciclos que possam existir na árvore de procura (estados repetidos).

Alguns dos critérios de classificação das estratégias de procura são [**?** **?**]:

- **Completa**: Se a estratégia encontra **sempre** uma solução para o problema proposto, caso exista (e caso não exista, diz que não há solução).
- **Ótima**: Se a estratégia encontra a solução com o menor custo possível.
- **Complexidade de espaço**: Espaço de memória necessário para encontrar a solução (i.e., número máximo de nós que a estratégia mantém em memória)
- **Complexidade de tempo**: Tempo necessário para encontrar a solução (i.e., número de operações necessárias para encontrar a solução)
- **Profundidade Máxima**: Corresponde à quantidade máxima de espaços entre um qualquer par de estados, ou seja, a profundidade máxima (m) da árvore de procura.
- **Fator de Ramificação**: Número máximo de sucessores de um dado nó (b).
- **Profundidade da solução**: Profundidade do nó objetivo na árvore de procura (d).

5.3.1 Procura em Grafo

Associado a uma estratégia de procura, podem estar associado o aparecimento de estados repetidos. Tal situação leva ao desperdício de recursos computacionais (tempo e memória), uma vez que o mesmo estado é gerado e expandido várias vezes.

A solução passa por manter em memória duas listas: uma com os nós que já foram expandidos (nós fechados); e outra com os nós gerados que ainda estão por expandir (nós abertos). No processo de memorização dos nós, associada a este processo de procura (ver secção 5.2.1), se o nó sucessor já existir na lista de nós abertos ou fechados, é necessário verificar se o mesmo foi atingido através de um caminho mais curto (com menor custo). Se for o caso, remover o nó anterior da respetiva lista e inserir o nó sucessor na lista de nós abertos. Se o nó sucessor não existir em nenhuma das listas, é apenas necessário inserir o nó na lista de nós abertos.

De forma a simplificar este processamento, pode ser criada uma única lista de nós (nós explo-

rados), onde são englobados os nós abertos e fechados. A lista deve ser indexada pelo estado do nó para eficiência de acesso, devido à elevada quantidade de nós que podem existir [**?**].

5.3.2 Procuras Não-Informadas

As procuras não-informadas (ver figura 5.3), são estratégias de procura que não utilizam informação adicional para guiar o processo de procura. Sabem apenas o que a definição do problema lhes transmite, sem qualquer tipo de pista ou heurística que permita saber se uma ação é "mais promissora" que outra [**?**]. Caracterizam uma exploração **exaustiva** do espaço de estados.

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

b – factor de ramificação
d – dimensão da solução
m – profundidade da árvore de procura
l – limite de profundidade
C^* – Custo da solução óptima
ε – Custo mínimo de uma transição de estado ($\varepsilon > 0$)

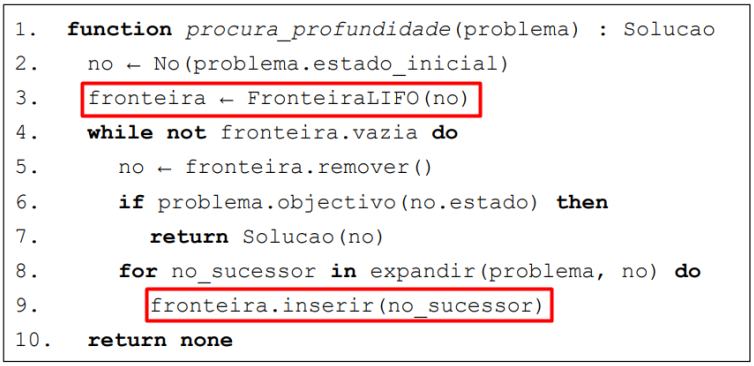
Figura 5.3: Tabela de estratégias de procura não-informadas. Retirado de [**?**], slide 12.

Procura em Profundidade

A procura em profundidade, também conhecida por *Depth-First Search* (DFS), é um mecanismo de procura caracterizado por explorar um ramo da árvore de procura até ao fim antes de explorar outro ramo. Utiliza uma fronteira de exploração do tipo *LIFO* (Last In First Out), onde os nós mais recentes são inseridos no início (de modo a serem expandidos primeiro). Este mecanismo não garante uma solução completa (visto que podem haver profundidades infinitas ou até mesmo ciclos) e muito menos ótima (já que o primeiro nó objetivo encontrado, se existir, é o que é retornado e que não é garantidamente o melhor).

A memorização dos nós, descrita no processo de procura associado (ver secção 5.2.1) a este mecanismo, é realizada apenas através da inserção dos nós na fronteira de exploração.

O algoritmo de procura em profundidade é ilustrado na figura 5.4, onde os retângulos a vermelho representam a memorização dos nós que é efetuada (ver secção 5.2.1) no respetivo processo de procura..



```
1.  function procura_profundidade(problema) : Solucao
2.    no ← No(problema.estado_inicial)
3.    fronteira ← FronteiraLIFO(no)
4.    while not fronteira.vazia do
5.      no ← fronteira.remover()
6.      if problema.objectivo(no.estado) then
7.        return Solucao(no)
8.      for no_sucessor in expandir(problema, no) do
9.        fronteira.inserir(no_sucessor)
10.   return none
```

Figura 5.4: Algoritmo de procura em profundidade. Retirado de [**?**], slide 15.

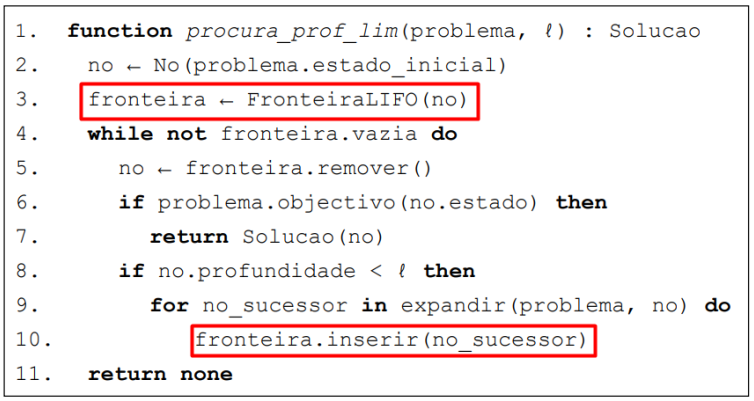
De notar que, a ordem de geração dos nós sucessores depende da ordem dos operadores aplicados [**?**] e que a complexidade espacial deste algoritmo é de $O(bm)$ se for implementado de forma recursiva, e de $O(m)$ se for implementado de forma iterativa.

Procura em Profundidade Limitada

A procura em profundidade limitada, é uma variação da procura em profundidade, onde é definido um limite de profundidade, l, para a exploração da árvore de procura. Com esta procura introduz-se um novo possível estado, o estado de fracasso, ou seja, se a profundidade máxima for atingida e não existir um nó objetivo, a procura termina sem solução.

Não é completa e garantidamente ótima, visto que a solução pode estar a uma profundidade maior que l, e que apesar de esta estratégia resolver o problema de profundidades infinitas, não resolve o problema da existência de ciclos.

O algoritmo de procura em profundidade limitada é ilustrado na figura 5.5, onde os retângulos a vermelho representam a memorização dos nós que é efetuada (ver secção 5.2.1) no respetivo processo de procura..



```
1.  function procura_prof_lim(problema, ℓ) : Solucao
2.    no ← No(problema.estado_inicial)
3.    fronteira ← FronteiraLIFO(no)
4.    while not fronteira.vazia do
5.      no ← fronteira.remover()
6.      if problema.objectivo(no.estado) then
7.        return Solucao(no)
8.      if no.profundidade < ℓ then
9.        for no_sucessor in expandir(problema, no) do
10.         fronteira.inserir(no_sucessor)
11.   return none
```

Figura 5.5: Algoritmo de procura em profundidade limitada. Retirado de [**?**], slide 17.

Procura em Profundidade Iterativa

A procura em profundidade iterativa, é uma variação da procura em profundidade limitada, onde é realizada uma procura em profundidade com um limite de profundidade, l. Caso não seja encontrada uma solução, é realizada uma nova procura em profundidade com um limite de profundidade superior ao que foi utilizado anteriormente, e assim sucessivamente até que a solução seja encontrada (se existir).

Combina vantagens da procura em profundidade e da procura em largura, visto que a procura em profundidade é mais eficiente em termos de memória e a procura em largura é mais eficiente em termos de tempo.

É completa, caso a profundidade seja finita, e ótima, caso o custo de cada ação seja maior que zero [**?**]. No entanto, se incremento de profundidade for maior que 1, deixa de ser uma procura ótima.

Procura em Largura

A procura em largura, também conhecida por *Breadth-First Search* (BFS), é um mecanismo de procura em grafo (ver secção 5.3.1) caracterizado por explorar todos os nós de um nível da árvore de procura antes de explorar os nós do nível seguinte. Utiliza uma fronteira de exploração do tipo *FIFO* (First In First Out). Neste tipo de fronteira, os nós mais recentes são inseridos no fim (de modo a serem expandidos por último).

Este mecanismo garante uma solução completa, se o fator de ramificação (b) for finito, e ótima, se o custo do caminho não diminuir ao aumentar a profundidade da árvore de procura [**?**].

O teste do nó objetivo deve ser realizado assim que o nó é gerado, para evitar processamento desnecessário (ver figura 5.6)., já que todos os nós que existem a uma profundidade menor (com menor custo) já foram explorados e não passaram no teste. Tal permite obter descer a complexidade temporal e espacial de $O(b^d + 1)$ para $O(b^d)$, visto que um nível da árvore de procura não é explorado a mais, nem são gerados nós desnecessários nesse nível [**?**].

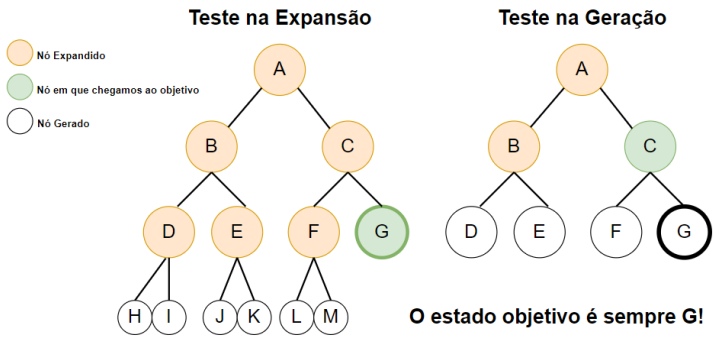


Figura 5.6: Comparação entre o teste do nó objetivo antes e depois da expansão dos nós sucessores. Retirado de [**?**].

O algoritmo de procura em largura é ilustrado na figura 5.7, onde os retângulos a vermelho representam a memorização dos nós que é efetuada (ver secção 5.2.1) no respetivo processo de procura. no respetivo processo de procura.

```
1.  function procura_largura(problema) : Solucao
2.    no ← No(problema.estado_inicial)
3.    fronteira ← FronteiraFIFO(no)
4.    explorados ← {no.estado : no}
5.    while not fronteira.vazia do
6.       no ← fronteira.remover()
7.       if problema.objectivo(no.estado) then
8.          return Solucao(no)
9.       for no_sucessor in expandir(problema, no) do
10.         estado ← no_sucessor.estado
11.         if estado not in explorados then
12.            explorados[estado] ← no_sucessor
13.            fronteira.inserir(no_sucessor)
14.   return none
```

Figura 5.7: Algoritmo de procura em largura. Retirado de [**?**], slide 28.

Procura de Custo Uniforme

A procura de custo uniforme, corresponde a um caso particular da procura por melhor primeiro (ver secção 5.3.3), onde a função de avaliação, $f(n)$, é igual ao custo do caminho percorrido desde o estado inicial até ao nó, ou seja, $f(n) = g(n)$. Portanto, o avaliador associado a esta procura é exclusivamente a função de custo, $g(n)$.

Como a procura de custo uniforme é também uma procura por grafo (ver secção 5.3.1), a memorização dos nós só é realizada se os nós sucessores não estiverem na lista de nós explorados ou se o custo do nó for menor que o custo do nó explorado (ver figura 5.9).

5.3.3 Procuras Informadas

As procuras informadas são estratégias de procura que utilizam informação adicional para guiar o processo de procura. Caracterizam uma exploração **seletiva** do espaço de estados.

Este tipo de procuras, baseiam-se, de uma forma geral, na estratégia de procura de melhor primeiro (ver secção 5.3.3). Esta estratégia, recorre a uma função de avaliação, $f(n)$, para escolher a ordem de expansão dos nós. É o avalidor que determina o valor de $f(n)$ de um determinado nó, e que vai depender da estratégia de procura utilizada.

A função de avaliação, $f(n)$, é modelada como uma estimativa do custo entre um nó n, e o objetivo. A figura 5.8 ilustra a função de avaliação de um procura informada. Poderá ser constituída por alguns dos seguintes componentes, através da transformação destes (soma/composição/supressão, etc..) [**?**]:

- $g(n)$, o custo do caminho percorrido desde o estado inicial até ao nó n;
- $h(n)$, a estimativa do custo do melhor caminho desde o nó n até ao nó objetivo, e que corresponde ao conceito de função heurística. Num nó objetivo, $h(n)$ é igual a zero;
- $h^*(n)$, o custo real do melhor caminho desde o nó n até ao nó objetivo, isto é, a estimativa exata.

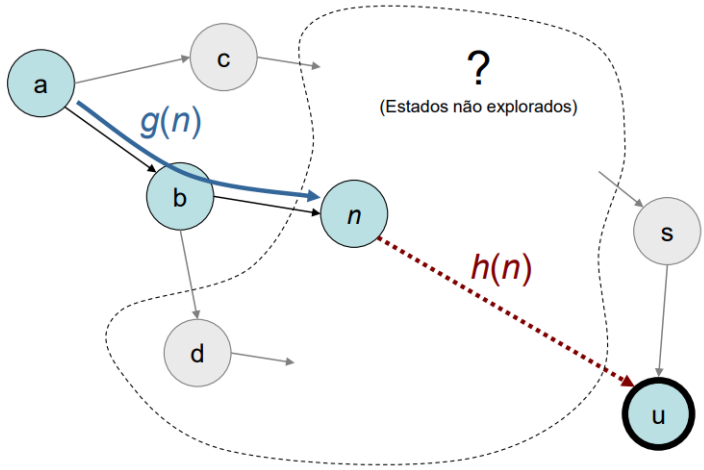


Figura 5.8: Ilustração dos componentes da função de avaliação de um procura informada. Retirado de [**?**], slide 5.

Heurística

Uma heurística é algo que fornece informação para resolver algo de forma expedita. Caracterizam-se por serem estimativas rápidas e por isso não são garantidamente ótimas, não produzem indicações robustas que garantidamente levam a boas soluções, no entanto, são boas indicações e ajudam a orientar a procura. Uma analogia seria, por exemplo, um detetor de metais, que apita cada vez mais alto à medida que se aproxima de um objeto metálico, mas não indicando, portanto, a sua localização exata [**?**].

Procura Melhor-Primeiro

A procura de melhor primeiro, também conhecida por *Best-First Search* (BFS), é um mecanismo de procura por grafo caracterizado por expandir o nó que tem o menor valor de uma função de avaliação, $f(n)$, que é uma estimativa do custo do caminho mais curto desde o nó até ao nó objetivo.

A função de avaliação, $f(n)$, representa a prioridade de um nó, e é calculada através da soma de duas funções: $g(n)$, que representa o custo do caminho percorrido desde o estado inicial até ao nó n; e $h(n)$, que representa a estimativa do custo do melhor caminho desde o nó n até ao nó objetivo.

Utiliza uma fronteira por prioridade, que internamente usa uma *min priority queue*, onde os nós são ordenados de acordo com o valor da função de avaliação, $f(n)$, de forma crescente. Desta forma, o nó com menor valor de $f(n)$ é o primeiro a ser expandido.

A procura de melhor primeiro é garantidamente completa e ótima, se a função de avaliação, $f(n)$, for admissível, ou seja, se a função heurística, $h(n)$, for menor ou igual ao custo real, $h^*(n)$, do melhor caminho desde o nó n até ao nó objetivo [**?**].

O algoritmo de procura de melhor primeiro é ilustrado na figura 5.9, onde os retângulos a vermelho representam a memorização dos nós que é efetuada (ver secção 5.2.1) no respetivo processo de procura.

```
1. function procura_melhor_prim(problema, f) : Solucao
2.   no ← No(problema.estado_inicial)
3.   fronteira ← FronteiraPrioridade(no, f)
4.   explorados ← {no.estado: no}
5.   while not fronteira.vazia do
6.     no ← fronteira.remover()
7.     if problema.objectivo(no.estado) then
8.       return Solucao(no)
9.     for no_sucessor in expandir(problema, no) do
10.      estado ← no_sucessor.estado
11.      if estado not in explorados or
12.         no_sucessor.custo < explorados[estado].custo then
13.        explorados[estado] ← no_sucessor
14.        fronteira.inserir(no_sucessor)
15. return none
```

Figura 5.9: Algoritmo de procura de melhor primeiro. Retirado de [**?**], slide 4.

Procura Sôfrega

A procura sôfrega, também conhecida por procura gananciosa (*Greedy Search*), é um mecanismo de procura por melhor primeiro, caracterizado por expandir o nó que tem o menor valor de uma função de avaliação, $f(n)$. O avaliador, associado a esta procura e que determina a prioridade do nó, é exclusivamente a função heurística, $h(n)$, que representa a estimativa do custo do melhor caminho desde o nó n até ao nó objetivo.

Esta estratégia tenta minimizar a estimativa de custo para atingir o objetivo não tendo em conta o percurso já explorado (só olha para a frente). Desta forma produz soluções subótimas, porque a função de heurística pode não ser admissível, ou seja, $h(n)$ pode ser maior que o custo real, $h^*(n)$.

Procura A*

A procura A*, é um mecanismo de procura por melhor primeiro, caracterizado por expandir o nó que tem o menor valor de uma função de avaliação, $f(n)$.

O avaliador, associado a esta procura e que determina a prioridade do nó, é a soma de duas funções: $g(n)$, que representa o custo do caminho percorrido desde o estado inicial até ao nó n; e $h(n)$, que representa a estimativa do custo do melhor caminho desde o nó n até ao nó objetivo ($f(n) = g(n) + h(n)$).

A procura A* é garantidamente ótima, se a função de avaliação, $f(n)$, for admissível, ou seja, se $h(n) <= h^*(n)$. Isto numa procura em árvore, porque numa procura em grafo, como não são adicionados à fronteira estados por onde já passámos, podemos eventualmente perder o caminho mais curto até ao objetivo. Este ponto, introduz um novo conceito de heurística, a heurística consistente. Uma heurística é consistente se, para cada nó n e cada sucessor n' de n gerado por qualquer ação a, o custo estimado de alcançar o objetivo a partir de n é no máximo o custo de chegar a n' mais o custo estimado de alcançar o objetivo a partir de n'. Mais concretamente se, $h(n) <= c(n, a, n') + h(n')$, onde $c(n, a, n')$ corresponde ao custo associado a realizar a ação n para n' [**?**]. Esta desigualdade triangular está representada, de forma visual, na figura 5.10.

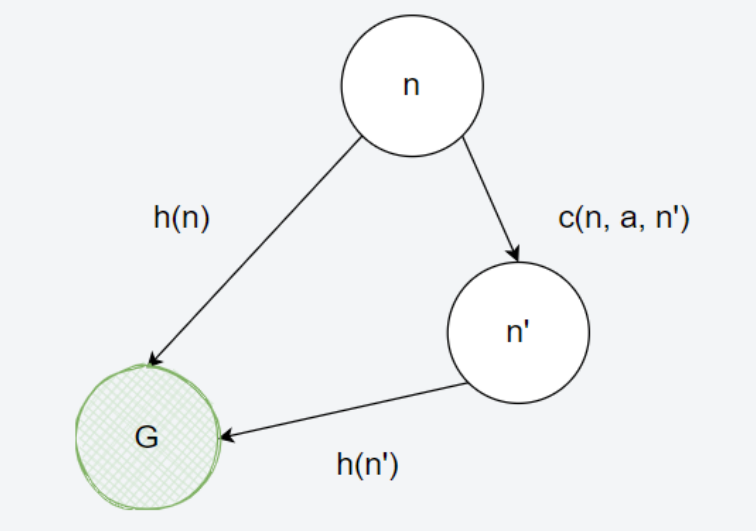


Figura 5.10: Desigualdade triangular para uma heurística consistente. Retirado de [**?**].

Portanto, uma heurística admissível pode não ser uma heurística consistente, visto que o seu valor pode variar de forma não linear ao longo do processo de procura, e por isso, não respeitar a desigualdade triangular mencionada anteriormente. E por isso, não há garantia de que um nó mais recente seja melhor que nós anteriores, ou seja, que esse nó esteja necessariamente no caminho ótimo. Desta forma, é necessário manter os nós anteriormente explorados para comparação [**?**].

De notar que, uma heurística consistente é sempre admissível, visto que respeita a desigualdade triangular, no entanto, o contrário não é verdade.

Na figura 5.11 é ilustrada a solução encontrada pela procura A* para um determinado problema, quando comparada com a procura de custo uniforme e a procura sôfrega.

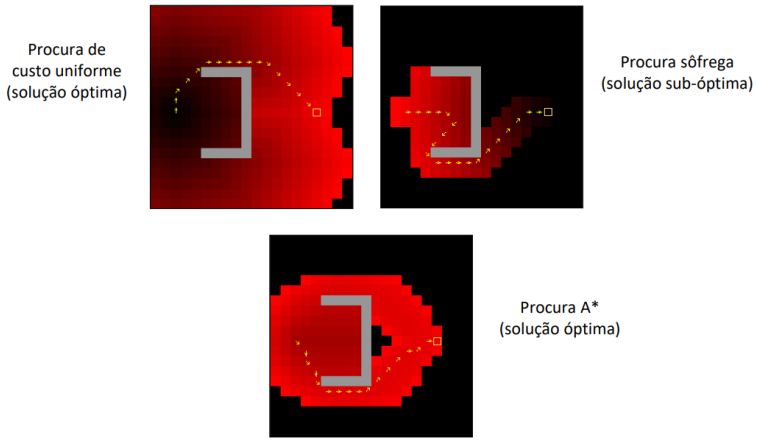


Figura 5.11: Comparação entre a procura A*, a procura de custo uniforme e a procura sôfrega. Retirado de [**?**], slide 13.

5.4 Aplicação das Estratégias de Procura

De forma a aplicar as estratégias de procura mencionadas a algo concreto, foi modelado um problema de contagem usando os componentes descritos na secção 5.1.1.

O problema de contagem é caracterizado por um valor inicial, um valor final e um conjunto de incrementos possíveis, onde o objetivo é atingir ou superar o valor final a partir do valor inicial, através destes. Por exemplo, se: o valor inicial for 0, o valor final for 9 e os incrementos possíveis forem [1, 2], então a solução ótima deverá ser, por exemplo, os estados: [0, 2, 4, 6, 8], com os incrementos [2, 2, 2, 2, 1].

Para as procuras informadas, foi necessário definir uma heurística admissível que refletisse a distância entre valores, para o sistema poder optar por transições de estado que minimizam a distância ao estado objetivo.

Com o problema modelado foram aplicadas as estratégias de procura estudadas e analisados os resultados obtidos, nomeadamente a solução encontrada, o custo e a dimensão associada, bem como o número de nós processados e o número máximo de nós mantidos em memória (i.e., de forma a inferir a complexidade espacial). Os resultados obtidos estão representados na tabela 5.1.

Através dos resultados obtidos, é possível concluir que a procura de custo uniforme e a procura A* são as estratégias que encontram a solução ótima, sendo que a procura A* apresenta o maior balanceamento entre a complexidade temporal e espacial. As outras estratégias encontram soluções subótimas. É também possível observar que a procura sôfrega sacrifica

Tabela 5.1: Resultados da aplicação das estratégias de procura ao problema de contagem.

Mecanismo de Procura	**Solução**	**Custo**	**Máximo de Nós**	
			processados	**memória**
ProcuraProfundidade	[2, 2, 2, 2, 2]	20.0	6	6
ProcuraProfLim	[2, 2, 2, 2, 2]	20.0	6	6
ProcuraProfIter	[2, 2, 2, 2, 2]	20.0	63	6
ProcuraLargura	[1, 2, 2, 2, 2]	17.0	10	11
ProcuraCustoUnif	[1, 1, 1, 1, 1, 1, 1, 1]	9.0	16	11
ProcuraSofrega	[2, 2, 2, 2, 1]	17.0	6	11
ProcuraAA	[1, 1, 1, 1, 1, 1, 1, 1]	9.0	10	11

o custo da solução em prol da complexidade temporal (apresenta uma das soluções com o menor número de nós processados), sugerindo uma abordagem gulosa, como era expectável.

De forma a testar o possível aparecimento de ciclos nas estratégias de procura que não são em grafo, foi adicionado um terceiro operador à modelação do problema de contagem - o operador (-1). Com a adição deste operador, verificou-se que, nas estratégias de procura em profundidade, a procura é infinita, visto que existe a formação de ciclos (e.g., [-1, 1, 2, -1, 1, 2, -1, ...]). Já nas outras procuras, a adição deste operador não afetou o resultado obtido, visto que estas, por serem em grafo, evitam o aparecimento de ciclos.

Este exercício permitiu perceber que a escolha da estratégia de procura ideal dependerá de uma análise cuidada dos fatores associados ao problema em questão, de forma a garantir a resolução eficiente e eficaz do mesmo.

5.5 Estrutura do Projeto

No processo de desenvolvimento de software associado a esta fase do projeto, foi definida a seguinte estrutura em módulos e que está presente na pasta *iasa_ agente/src*:

- *agente/contagem*: Integra a modulação que foi feita para um determinado problema de contagem, bem como o ficheiro onde lhe são aplicadas as estratégias de procura estudadas;
- *lib/mod*: Contém classes que ajudam a modular um problema de forma abstrata (i.e., conceito de operador, estado)
- *lib/pee*: Contém sub-módulos que permitem modular as diferentes estratégias de procura em espaço de estados (pee) estudadas:
  - *largura*: Contém classes que representam a estratégia de procura em largura e os seus componentes (e.g., fronteira-fifo);
  - *profundidade*: Contém classes que representam a estratégia de procura em profun-

didade, os seus componentes (e.g., fronteira-lifo) e variantes (e.g., profundidade-limitada, profundidade-iterativa);

- *mec_proc*: Contém classes que representam o mecanismo de procura e os seus componentes (e.g., no, fronteira, solução)
- *melhor_prim*: Contém classes que representam as estratégia de procura de melhor primeiro (e.g., procura-aa, procura-custo-unif) e os seus componentes (e.g., avaliadores)

Capítulo 6

Projeto - Parte 4

TODO()

6.1 Arquitetura Deliberativa

Conforme foi abordado na secção 4.3, um agente reativo com memória age com conhecimento do presente, através de reações a estímulos, e com conhecimento do passado, através da memorização de percepções e de ações passadas. Um agente deliberativo, além de possuir as características de um agente reativo com memória, é capaz de antecipar o futuro, através da simulação de cenários, e consequentemente de planear ações futuras, com base em objetivos explícitos (fixos ou gerados dinamicamente) e em processos de deliberação sobre que objetivos concretizar e quais os meios a utilizar.

Qualquer sistema para poder antecipar o futuro tem que ter conhecimento (modelo do mundo), e esse conhecimento pode ser adquirido através da experiência ou de algo que já tenha esse conhecimento e o transmita para o agente (e.g., num ambiente multi-agente, um agente pode adquirir conhecimento de outro agente). O modelo do mundo caracteriza-se como um caso particular da representação do modelo do problema, e permite simular para cada opção as múltiplas sequências de evolução possíveis, através de uma simulação interna [**?**].

A sequência de ações geradas por um agente deliberativo, através de processos internos, caracterizam o seu comportamento, ao contrário dos agentes reativos que têm comportamentos baseados em reações.

Numa arquitetura deliberativa, o módulo de memória é indispensável e dá suporte à simulação interna e aos mecanismos de deliberação [**?**], conforme representado na figura 6.1.

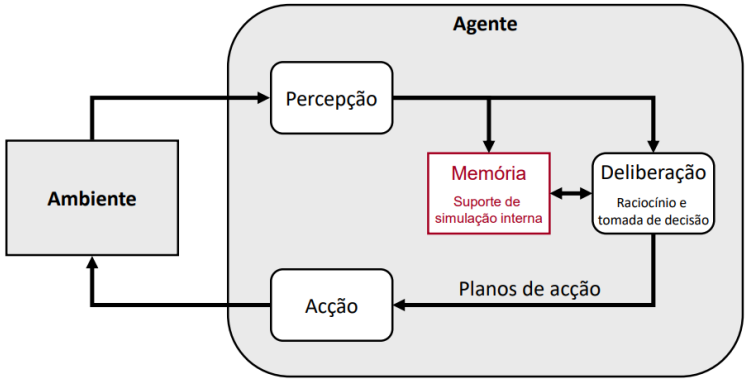


Figura 6.1: Arquitetura deliberativa. Retirado de [**?**], slide 16.

6.2 Raciocínio Automático

Conforme foi abordado na secção 5.1, o raciocínio automático é um processo de inferência que permite a um agente deliberativo deduzir novas informações a partir de conhecimento prévio.

Com base nos conhecimentos adquiridos nesta fase do projeto, este raciocínio pode ser subdividido em dois tipos:

- **Raciocínio Prático**: orientado para a ação (interação permanente com o mundo, ao qual está associado o processo de tomada de decisão). Tem como input os objetivos a atingir, as ações realizáveis e a representação do mundo, e como output os planos de ação.
- **Raciocínio Teórico**: orientado para o conhecimento, que permite deduzir novas informações a partir de conhecimento prévio. Caracteriza-se por ser direto, ou seja, não envolve interação com o mundo.

6.2.1 Raciocínio Prático

Numa arquitectura deliberativa, o raciocínio prático suporta o processo geral de tomada de decisão que determina o comportamento do agente, ou seja, quais as acções a realizar perante as percepções obtidas e o estado do modelo interno do mundo [**?**]. Este processo de tomada de decisão é composto por duas fases:

- **Deliberação**: raciocínio sobre os fins, que permite definir os objetivos a atingir (i.e., opções (input) -> objetivos (output)).
- **Planeamento**: raciocínio sobre os meios, que permite definir os planos de ação a executar (i.e., ações (input) -> planos (output)).

Tendo em conta que o raciocínio prático é um processo de tomada de decisão, é necessário que o agente tenha objetivos, que são os critérios de avaliação das ações, e que possa avaliar as

ações possíveis, de forma a escolher a melhor ação a executar. Portanto, a deliberação sobre a representação do modelo do mundo que permite definir os objetivos, e o planeamento com base nesses e nos meios disponíveis, definem os processos necessários para a geração de planos de ação, conforme representado na figura 6.2.

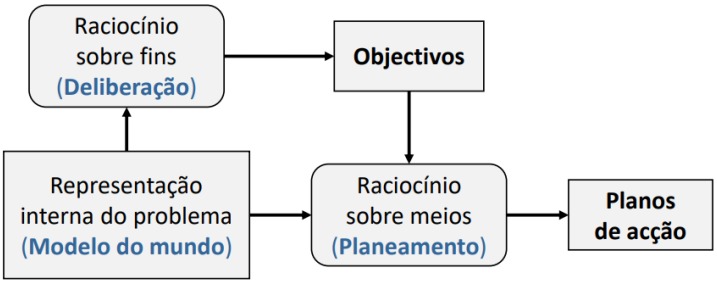


Figura 6.2: Deliberação e planeamento num agente deliberativo. Retirado de [**?**], slide 8.

No entanto, o ambiente pode ser dinâmico, e por isso o agente pode ter que reagir a mudanças no ambiente (i.e., alterações no modelo do mundo), o que implica que o processo de tomada de decisão seja adaptativo, ou seja, que possa ser reavaliado e ajustado (reconsideração). Além disso, mesmo sem alteração do estado do ambiente, o próprio plano de ação pode alterar-se ao longo do tempo e estar desfasado da realidade, devido a um acontecimento que não foi antecipado (e.g., um camião que seguia um trajeto específico, mas por causa de areia no caminho, teve que mudar de trajeto).

Portanto, o processo geral de tomada de decisão, que ocorre de forma cíclica, pode ser então representado pelos seguintes passos [**?**]:

1 Observar o mundo, gerando percepções.

2 Atualizar o modelo do mundo, com base nas percepções.

3 Se reconsiderar, então:

  3.1 Deliberar o que fazer, gerando um conjunto de objetivos.

  3.2 Planear como fazer, gerando um plano de ação.

4 Executar o plano de ação.

6.2.2 Racionalidade Limitada

A racionalidade limitada é um conceito que se aplica a agentes que têm recursos computacionais limitados (i.e., tempo de computação, memória), e que por isso não conseguem gerar planos de ação ótimos.

Um agente com racionalidade limitada tenta chegar a uma solução satisfatória, não à melhor

solução, de forma dinâmica, tendo em conta os recursos computacionais disponíveis. Divide o problema em subproblemas mais simples, aplicando restricções de forma a reduzir a complexidade do problema, e resolve-os sequencialmente, de forma a atingir o objetivo final [**?**].

Para o contexto humano, não é possível fazer raciocínio ótimo mas sim automático, devido à racionalidade limitada.

6.3 Planeamento Automático

O planeamento automático é um processo deliberativo, que faz parte do raciocínio prático, que tem por objetivo gerar sequências de ação, designadas planos [**?**]. Um processo de planeamento requer um modelo de planeamento que define a representação abstrata do problema a resolver, e que apresenta os mesmos conceitos que foram abordados na secção 5.1.1 (i.e., estados, operadores, etc...). Dado um modelo de planeamento e um conjunto de objetivos provenientes do processo de deliberação, o planeador gera um plano de ação que permite ao agente concretizar esses objetivos.

Este processo é realizado através de métodos de raciocínio automático, como a procura em espaço de estados e a procura por processos de decisão de Markov [**?**].

6.3.1 Planeador Baseado em PEE

No caso de um planeador baseado em procura em espaços de estados (PEE), é necessário considerar [**?**]:

- Modelo do problema de planeamento.
- Heurística a utilizar (se necessário) para guiar a procura (ver secção 5.3.3).
- Mecanismo de procura (ver secção 5.2).

6.3.2 Planeador Baseado em PDM

No caso de um planeador baseado em processos de decisão de Markov (PDM), a representação do domínio do problema é feita sobre a forma de um processo de decisão de Markov, que se define como um modelo matemático que descreve um sistema que evolui ao longo do tempo, com base em estados e ações, e que é utilizado para modelar situações de decisão sequencial sob incerteza [**?**].

Ao contrário do planeador baseado em PEE, o planeador baseado em PDM não gera um plano de ação, mas sim uma política. Esta política define-se como uma função que mapeia estados em ações, e que é utilizada para determinar a ação a executar em cada estado, definindo a estratégia de tomada de decisão do agente.

Além disso, em vez de só ter em conta o custo, o planeador baseado em PDM tem em conta a utilidade de uma ação que depende de uma sequência de decisões, da possibilidade de ganhos e perdas, da incerteza na decisão e do efeito cumulativo [?].

6.4 Processos de Decisão Sequencial

No processo de decisão geral, cada estado s é caracterizado por um conjunto de atributos que descrevem o estado do mundo, e cada ação a é uma transformação que altera o estado do mundo, levando a um novo estado s' [?]. O valor (utilidade) desses estados e decisões pode não ser conhecido de imediato, sendo percebido apenas de forma diferida no tempo, devido a potenciais encadeamentos de estados e possíveis decisões futuras, conforme representado na figura 6.3.

No processo de decisão sequencial, a evolução entre estados ocorre por efeito de ações que, no caso geral, podem ser não deterministas, ou seja, o resultado das ações pode não ser completamente previsível, podendo existir incerteza no seu resultado. Esse tipo de processos corresponde a espaços de estados (ambientes) não deterministas (estocásticos), nos quais as transições entre estados ocorrem por efeito das ações, cada uma com uma probabilidade associada de forma a representar a incerteza do ambiente. A cada transição pode estar associada uma recompensa, que representa o ganho ou perda relacionado a essa transição de estado [?].

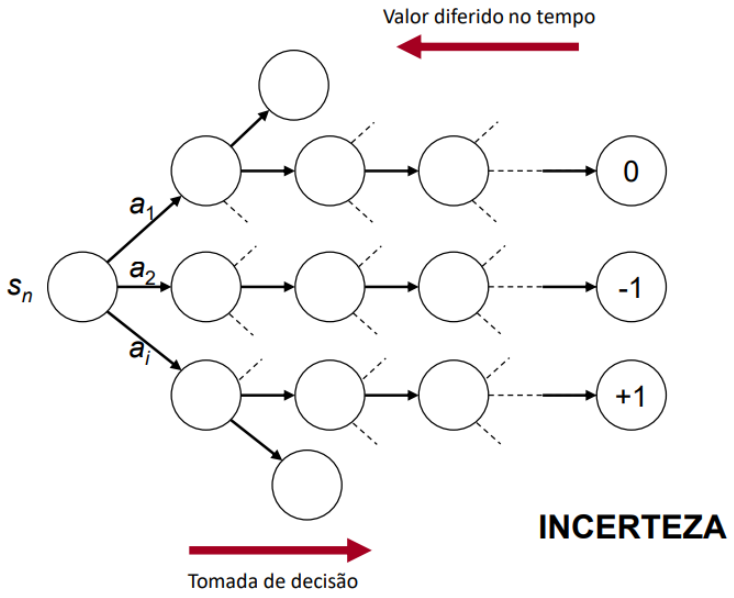


Figura 6.3: Processo de decisão sequencial. Retirado de [?], slide 3.

6.5 Cadeias de Markov

Uma cadeia de Markov [**?**] é um processo estocástico que evolui ao longo do tempo, com base em estados e transições entre esses estados, e que obedece à propriedade de Markov [**?**].

A propriedade de Markov é uma propriedade em processos estocásticos, que se caracteriza pela distribuição probabilística condicional dos estados futuros de um processo depender exclusivamente do estado presente (i.e., a previsão dos estados seguintes só depende do estado presente) [**?**].

Formalmente, uma cadeia de Markov é definida por um conjunto de estados S e por uma função de transição T, que define a probabilidade de transição de um estado para outro, conforme representado na figura 6.4.

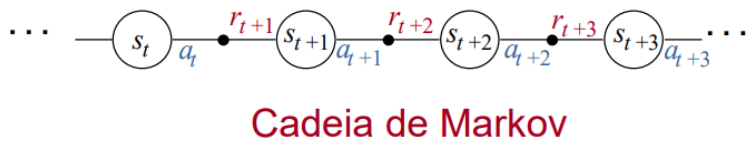


Figura 6.4: Cadeia de Markov. Retirado de [**?**], slide 7.

6.6 Processos de Decisão de Markov

O processo de decisão de Markov (PDM) é um processo de controlo estocástico em tempo discreto, e que fornece um modelo matemático para modelar situações de decisão sequencial sob incerteza [**?**]. Esta incerteza resulta da impossibilidade de se obter informação completa relativa ao domínio do problema (e.g., navegação em veículos autónomos). Este processo caracteriza-se como uma extensão de uma cadeia de Markov, que inclui, além dos estados e transições; ações e recompensas, de forma a obter uma política. Formalmente, é representado pelo seguinte conjunto de elementos:

- S: conjunto finito de estados;
- **Modelo de transição**: $T(s, a, s')$, onde s é o estado atual, a é a ação a ser tomada, e s' é o estado futuro. é a probabilidade de mudança de estado de s para s' após a ação a ser tomada. Em ambientes deterministas, a probabilidade de transição é 1, e em ambientes estocásticos, a probabilidade de transição é um valor entre 0 e 1;
- **Modelo de recompensa**: $R(s, a, s')$, onde s é o estado atual, a é a ação a ser tomada, e s' é o estado futuro. é a recompensa associada à transição de estado de s para s' após a ação a ser tomada. Este modelo está descrito no caso geral, mas poderá estar dependente apenas do estado atual e da ação tomada ou apenas do estado atual;
- **Factor de desconto**: γ, onde $0 \leq \gamma \leq 1$, que é o fator de retenção aplicado à recom-

pensa. Por cada unidade de tempo que passa, a recompensa é descontada por um fator de γ, de forma exponencial. Se $\gamma = 0$, a tomada de decisão só depende do presente, porque o futuro é anulado.

6.6.1 Utilidade

O valor (utilidade) de um estado caracteriza-se com o efeito acumulativo que vai tendo do mundo, através dos ganhos e perdas que vai obtendo, diferidos no tempo, visto que pode existir perda de oportunidade na passagem do tempo (e.g., no mercado financeiro, o valor de uma ação é volátil e varia ao longo do tempo). No entanto, poderão existir estados em que não existem ganhos e perdas.

Portanto, a utilidade representa o valor de se estar num estado, para efeito de concretização de um objetivo do sistema, mas representa o valor a longo prazo (na história de evolução do sistema) (e.g., um automóvel pode ter que acelarar, gastando mais combustivel, para evitar um acidente, mas a longo prazo, o valor de evitar o acidente é maior do que o valor de gastar mais combustivel).

6.6.2 Política

Uma política (π) é uma função que mapeia estados em ações, e que é utilizada para determinar a ação a executar em cada estado, definindo a estratégia de tomada de decisão do agente. É caracterizada em dois tipos:

- **Determinística**: mapeia cada estado em apenas uma ação ($\pi : S \rightarrow A(s); \quad s \in S$);
- **Estocástica**: mapeia cada estado numa distribuição de probabilidade sobre as ações possíveis ($\pi : S \times A(s) \rightarrow [0, 1]; \quad s \in S$).

6.6.3 Objetivo

O objetivo de um processo de decisão de Markov é determinar a política ótima (π^*), que é a política que maximiza a utilidade esperada, ou seja, a política que maximiza a recompensa esperada ao longo do tempo, através das equações de Bellman [**?**]. Devido à propriedade de Markov, a utilidade de um estado dada uma ação é a soma da recompensa imediata e da utilidade esperada do estado seguinte, ponderada pelo fator de desconto γ, conforme representado na equação de Bellman 6.1.

$$U(s, a) = \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma U^{\pi}(s')] \tag{6.1}$$

De forma a calcular a utilidade esperada de um estado, é necessário considerar a ação que maximiza a utilidade esperada desse estado, conforme representado na equação de Bellman 6.2.

$$Ufinal(s) = \max_a U(s, a) \tag{6.2}$$

A partir da utilidade esperada de cada estado, é possível determinar a política ótima (π^*), que é a política que seleciona a ação que maximiza a utilidade esperada de cada estado, conforme representado na equação de Bellman 6.3.

$$\pi^*(s) = \arg\max_a Ufinal(s, a) \tag{6.3}$$

Algoritmo de Iteração de Valor

O algoritmo de iteração de valor é um algoritmo que permite calcular a utilidade ótima de cada estado, através da iteração de cálculos de utilidade, até que a utilidade de cada estado convirja para um valor estável. Este algoritmo inicializa a utilidade de cada estado a zero e, de forma iterativa, calcula a utilidade de cada estado, conforme a equação de Bellman 6.2, até que um determinado critério de convergência seja atingido [**?**].

O critério de convergência é representado por um δ, que define a diferença máxima permitida entre a utilidade atual de um estado e a utilidade do estado na iteração anterior. À medida que as utilidades de cada estado são atualizadas ao longo das iterações, elas convergem para valores muito próximos, até que a partir de um determinado ponto são irrisórias, e consequentemente consideradas estáveis. Isto acontece porque a política converge mais cedo que a utilidade, por isso o δ é usado para verificar a convergência da política.

O algoritmo apresenta assim 2 ciclos: um ciclo externo que controla a convergência do algoritmo, e um ciclo interno que calcula a utilidade para cada estado, conforme ilustrado na figura 6.5.

```
function utilidade:
    U[s] ← 0, ∀s ∈ S
    do:
        U_ant ← U
        δ ← 0
        for s in S:
            U[s] ← max_{a∈A(s)} U_acção(s, a, U_ant)
            δ ← max{δ, |U[s] - U_ant[s]|}
    while δ > Δ_max:
    return U
```

Figura 6.5: Algoritmo de iteração de valor. Retirado de [**?**], slide 21.

Também de notar que o algoritmo de iteração de valor é um algoritmo de programação dinâmica, que é uma técnica de otimização que consiste em dividir um problema em subproblemas mais simples, guardando soluções intermédias, de forma a evitar o recalculo de soluções já calculadas.

6.7 Aprendizagem Automática

Num conceito geral, a aprendizagem define-se como a melhoria de desempenho, para uma dada tarefa, com a experiência. Esta experiência tem por base a memória, que é o grau mais básico da inteligência. No entanto, aprender não é apenas memorizar; é ganhar a capacidade de adaptação.

A aprendizagem automática define-se como um campo da inteligência artificial que se foca no desenvolvimento de algoritmos e técnicas que permitem aos sistemas aprender a partir de dados, de forma a melhorar o seu desempenho em tarefas específicas. Esta aprendizagem tem como base os seguintes elementos [**?**]:

- **Qual é a tarefa a ser aprendida (T)**: é a tarefa que o sistema tem que aprender, e que é definida pelo problema a resolver (e.g., jogar xadrez).
- **Métrica de desempenho (D)**: é a métrica que permite avaliar o desempenho do sistema, e que é utilizada para medir a qualidade da solução obtida (e.g., número de jogos ganhos)
- **Com base na experiência, como é que o sistema vai aprender (E)**: é o processo de aprendizagem, que é a forma como o sistema adquire conhecimento (e.g., jogos realizados).

Por exemplo, nas arquiteturas reativas, a aprendizagem é implícita nas associações estímulo-resposta (aprendizagem por associação), e nas arquiteturas deliberativas, a aprendizagem é explícita, através da simulação de cenários e do planeamento de ações futuras.

A aprendizagem automática divide-se em dois tipos [**?**].

- **Conceptual**: Refere-se ao que o sistema aprende (e.g., aprendizagem supervisionada, aprendizagem não supervisionada);
- **Comportamental**: Refere-se a como o sistema aprende (e.g., aprendizagem por reforço).

6.7.1 Aprendizagem por Reforço

A aprendizagem por reforço é uma técnica de aprendizagem automática que treina software para tomar decisões de modo a obter os melhores resultados, imitando o processo de aprendi-

zagem por tentativa e erro que os humanos utilizam para atingir os seus objetivos. Neste tipo de aprendizagem, as ações que contribuem para o objetivo são reforçadas (i.e., recompensadas positivamente), enquanto as ações que prejudicam o objetivo são ignoradas (ou punidas) (i.e., recompensadas negativamente) [**?**].

Em processos de decisão de Markov, este tipo de aprendizagem é utilizado para ajudar os sistemas de inteligência artificial a obter resultados ótimos em ambientes parcialmente observáveis, estocásticos, dinâmicos ou até contínuos, onde a modelação do mundo é complexa e/ou desconhecida. Por exemplo, a dimensionabilidade dos espaços de estados pode ser muito grande, a definição de modelos do mundo pode ser difícil tendo por base dados experimentais ou amostras, e os modelos de transição e de recompensa podem ser desconhecidos [**?**]. Por este motivo, o modelo precisa de ser construído a partir da experiência de forma incremental, nomeadamente através da tentativa e erro, com o objetivo de ajustar a política de decisão e maximizar a utilidade esperada.

Aprendizagem por Valor de Ação

A aprendizagem por valor de ação é um técnica de aprendizagem por reforço que tem por objetivo aprender a função que estima o valor de cada ação ($Q(a)$) (e.g., decisão entre várias ações com valores desconhecidos), de forma a maximizar a recompensa a longo prazo. Esta função pode ser determinada, por exemplo, através: da escolha repetida de diferentes ações (i.e., qual o valor médio para uma ação após n tentativas); da recompensa obtida por escolher uma determinada ação, de acordo com uma distribuição de probabilidades [**?**].

No entanto, a aprendizagem por valor de ação tem a desvantagem de se ter que realizar n tentativas para cada ação e manter informação sobre os resultados obtidos, de forma a calcular a estimativa de valor de cada ação. Tal abordagem pode ser ineficiente do ponto de vista de recursos computacionais, e não permitir aproveitar o conhecimento obtido de forma incremental.

Além disso, a recompensa pode variar ao longo do tempo (em distribuições não estacionárias) e por isso é necessário ajustar a função de valor de ação de forma a refletir essa variação. A taxa de aprendizagem (α) é um fator de ajuste que é aplicado ao erro de previsão, para ajustar a previsão de forma exponencial por cada unidade de tempo que passa, e que regula a importância das informações mais recentes em relação às informações mais antigas.

- Se $\alpha = 0$, o sistema não aprende, porque o erro de previsão é anulado (i.e., maior relevância das recompensas mais antigas);
- Se $\alpha = 1$, o sistema aprende, porque o erro de previsão é maximizado (i.e., torna-se um agente reativo, é guiado pela recompensa imediata, maior relevância das recompensas mais recentes) [**?**].

Portanto, se o agente apenas escolher a ação que atualmente tem a maior estimativa de valor (abordagem Greedy), pode ficar preso a uma política subótima devido a máximos ou mínimos locais, sem explorar o ambiente para obter mais conhecimento.

Existe, portanto, um compromisso (trade-off) entre explorar (para obter conhecimento) e aproveitar (o conhecimento adquirido até ao momento, para maximizar a recompensa). Uma abordagem possível para balancear este compromisso é a seleção de ações com base na política ϵ-Greedy. Nesta estratégia, o agente escolhe a ação com maior valor estimado com probabilidade $1 - \epsilon$, e escolhe uma ação aleatória com probabilidade ϵ [**?**]. A ideia dessa estratégia é que, ao explorar ocasionalmente, o agente pode descobrir ações melhores do que aquelas que inicialmente acreditava serem as melhores num determinado momento, o que contribui para a melhoria da política de decisão.

O fator de exploração (ϵ) regula o compromisso entre explorar e aproveitar, permitindo que, em determinados momentos, o agente explore o ambiente para obter conhecimento, e, em outros momentos, utilize o conhecimento adquirido até então para agir de forma a maximizar a recompensa. Para convergir para o valor ótimo, um agente não pode apenas explorar nem apenas aproveitar, mas sim encontrar um equilíbrio entre os dois. No entanto, é recomendável que a exploração seja progressivamente reduzida ao longo do tempo [**?**].

Apredizagem por Diferença Temporal

A aprendizagem por diferença temporal [**?**] é uma técnica de aprendizagem por reforço que combina elementos de métodos de Monte Carlo e programação dinâmica. Esta técnica tem por objetivo estimar o valor por acumulação de recompensas de forma não linear para lidar com ambientes não estacionários (regulada pelo fator de aprendizagem $\alpha \in [0, 1]$), e atualizar a estimativa de valor de estado-acção com base na sua mudança (diferença temporal) entre instantes sucessivos [**?**].

A aplicação prática da aprendizagem por diferença temporal pode ser feita através de dois tipos de algoritmos [**?**]:

- **Off-policy**: Atualiza a estimativa de valor de estado-ação com base na ação que maximiza o valor do estado subsequente, independentemente da ação escolhida pelo agente (por exemplo, o algoritmo Q-Learning). Neste algoritmo, a política de escolha de ação, como a estratégia ϵ-Greedy para exploração, determina como o agente explora o ambiente enquanto aprende. No entanto, a atualização dos valores de estado-ação segue uma abordagem Greedy, utilizando a melhor estimativa possível para a próxima ação.
- **On-policy**: Atualiza a estimativa de valor de estado-ação com base nas ações que o agente efetivamente executa de acordo com a política atual (aprendizagem incremental a partir da experiência) (por exemplo, algoritmo SARSA - State-Action-Reward-

State-Action). Para resolver o dilema entre explorar e aproveitar, a seleção de ação no algoritmo SARSA pode ser implementada com a estratégia ϵ-Greedy. Isso significa que o agente escolhe ações de forma ϵ-Greedy enquanto continua a melhorar as suas estimativas de valor de estado-ação com base nas ações reais que executa.

6.8 Implementação do Controlo Deliberativo

6.8.1 Implementação do Agente Deliberativo com PEE

6.8.2 Implementação do Agente Deliberativo com PDM

TODO()

Capítulo 7

Revisão do Projeto

Neste capítulo, são identificados e descritos os erros cometidos nas entregas realizadas, com a indicação do problema e da respectiva correção, justificada com base nos conhecimentos adquiridos.

7.1 Problema de Contagem

De forma a complementar o problema de contagem com toda a informação necessária para comparar a aplicação das diferentes estratégias de procura estudadas, foi necessário adicionar contadores que permitissem medir número máximo de nós em memória e o número de nós expandidos, correspondente à complexidade espacial e temporal, respetivamente.

Como todas as estratégias de procura implementadas têm como base um mecanismo de procura, foram acrescentados métodos com uma implementação base que permitissem a contagem dos nós expandidos e em memória. Enquanto que, para realizar a contagem de nós expandidos foi necessário manter um contador de nós expandidos que é incrementado por cada nó que é removido da fronteira para processamento; para realizar a contagem de nós em memória foi necessário manter informação acerca do número máximo de nós na fronteira de exploração. No entanto, para as procuras em grafo que mantêm informação de nós explorados, foi redefinido o método de obtenção de nós em memória para que fosse o número de nós explorados, pois já englobam os nós na fronteira de exploração.

Na procura iterativa em profundidade, foi necessário adicionar um acumulador de nós processados, visto que esta procura é uma procura em profundidade que é executada várias vezes, sendo necessário manter a contagem de nós processados em cada execução. Foi redefinido o método de obtenção de nós processados para que fosse o resultado do acumulador de nós processados.

7.2 Procura Iterativa em Profundidade

Durante a comparação das diferentes estratégias de procura, foi detetado que a procura iterativa em profundidade não estava a ser executada corretamente. A profundidade máxima de procura permanecia sempre no valor default e não era alterada dinamicamente através do setter disponível para o efeito. Este problema deveu-se a um erro ortográfico na variável associada ao setter, o que impediu a sua chamada.

Infelizmente, este erro passou despercebido porque o Python não gerou um erro, ao contrário do que ocorreria em Java, por exemplo, que geraria um erro de compilação. Além disso, o ciclo de procura não começava com a profundidade em zero, mas sim em um, o que contribuiu para a execução incorreta do algoritmo.

7.3 Agente Deliberativo com PDM

Na última aula prática não foi possível concluir a implementação do agente deliberativo para a procura com processos de decisão de Markov (PDM), devido a um problema que não foi detetado aquando da implementação do agente deliberativo na procura por espaço de estados (PEE).

Ambos utilizam o controlo deliberativo que apresenta um módulo que corresponde às representações internas do mundo - modelo do mundo. Este módulo que apresenta um método para obter os operadores, estava a criar novas cópias dos operadores disponíveis a cada chamada, o que estava incorreto, pois os operadores devem ser sempre os mesmos. Como tal, foi corrigido o método para que devolvesse sempre a mesma lista de operadores, inicializada no construtor do modelo do mundo. Este problema não foi descoberto na implementação do agente deliberativo para a procura por espaço de estados, pois o agente apenas constroi um plano de ação por cada reflexão, enquanto que na procura com processos de decisão de Markov, o agente tem de refletir várias vezes para ter uma política ótima, o que levou a que fosse criada uma nova lista de operadores a cada reflexão.

Ainda no módulo do agente deliberativo, quando usadas simulações do ambiente mais complexas, o agente deliberativo não conseguia convergir para uma política ótima, devido à falta de exploração do espaço de estados (i.e., logo de início, o agente não se mexia para explorar o ambiente). De forma a corrigir este comportamento, foi alterado o gama no `PlaneadorPDM` para um valor mais próximo de 1, visto que é o factor de desconto que controla a importância de recompensas futuras, e, consequentemente, a exploração num determinado momento em prol da ação com o conhecimento atual.

7.4 Comentários

Na revisão do projeto foram encontrados os seguintes módulos sem comentários por esquecimento, que foram corrigidos com a adição de comentários que descrevem o propósito de cada método e a sua implementação:

- `PlanoPDM`;
- `PlaneadorPDM`;
- `MecUtil`;
- `PDM`;

Capítulo 8

Conclusão

O projeto permitiu adquirir conhecimentos sobre inteligência artificial e os mecanismos base necessários para a implementação de agentes autónomos, através da aplicação prática de conceitos estudados ao longo do semestre.

Outras temáticas, como a arquitetura de software e o processo de desenvolvimento de software, foram também abordadas, aprendidas e aplicadas ao longo do projeto, o que permitiu a aquisição de competências em áreas como a análise, o design e a implementação de sistemas de software.

Foram adquiridas competências de programação em *java* e *python*, especialmente vocacionadas para a programação orientada a objetos, o que permitiu modular, de forma incremental, os diferentes subsistemas das arquiteturas implementadas, reduzindo a complexidade inerente ao desenvolvimento de sistemas de software.

Em relação as diferentes partes do projeto, na primeira parte, foi desenvolvida uma biblioteca em *java* onde foram implementados os mecanismos bases relacionados com os conceitos gerais de inteligência artificial (e.g., agente, ambiente) e um jogo que integra essa biblioteca. O jogo consistia num ambiente virtual onde a personagem tinha por objectivo registar a presença de animais através de fotografias, simulando a exploração no mundo real. O agente foi implementado com uma arquitetura reativa com memória, mais concretamente, criou-se uma máquina de estados finita no seu módulo de controlo, que permitia a tomada de decisões com base no estado anterior, em eventos pre-definidos do ambiente e na ação a executar pelo agente. Com esta arquitetura observou-se que o agente conseguia atingir os objetivos propostos, mas não conseguia reagir a situações inesperadas (i.e., eventos não previstos que pudessem ocorrer no ambiente). Mais ainda, na caracterização do ambiente, considerou-se que o mesmo era dinâmico e parcialmente observável, o que permite concluir que a arquitetura reativa implementada não é a mais adequada para o problema proposto, visto que não consegue reagir a situações inesperadas.

Na segunda parte do projeto, foi desenvolvido um agente reativo em *python* com módulos comportamentais para recolher alvos e evitar obstáculos num ambiente virtual com dimensões fixas. O agente foi implementado com uma arquitetura reativa simples (i.e., sem memória), onde um comportamento composto (que englobava os subcomportamentos: aproximar alvo, evitar obstáculo, explorar) foi integrado no módulo de controlo. Este módulo comportamental ficou responsável por processar a informação sensorial do agente e escolher a ação a executar com base nessa informação. Esta integração permitiu a escolha da ação a executar com base num mecanismo de seleção de ação, mais concretamente, por hierarquia fixa de prioridade. Com esta arquitetura observou-se, através da interface gráfica da biblioteca SAE, que o agente conseguia atingir os objetivos propostos, mas não de forma ótima (i.e., o caminho mais rápido para recolher todos os alvos, evitando os obstáculos, não era seguido).

Após análise dos resultados obtidos, concluiu-se que nenhum dos agentes implementados pode ser considerado um agente relacional (ver secção 2.3), uma vez que não conseguem realizar as ações corretas de forma ótima, apenas conseguem atingir os objetivos propostos. Visto que ambas as arquiteturas implementadas são arquiteturas reativas, a resposta pode estar nas arquiteturas deliberativas e híbridas (ver secção 2.4), que não foram abordadas neste projeto.