# Introdução à Inteligência Artificial

## Resumo (Teórica)

2021-2022

# Índice

## Tipos de ambiente

- Acessível
- Determinístico
- Episódico
- Dinâmico
- Discreto

## Ambiente Acessíveis/Não acessíveis

Se o conjunto de sensores do agente lhe der **acesso ao estado completo do ambiente**. Caso contrário diz-se **não acessível**.

## Ambientes Determinísticos/Estocásticos

O ambiente é **determinístico** se o seu próximo estado puder ser **completamente determinado** a partir do seu estado atual e da ação a executar, caso contrário diz-se **estocástico**.

## Ambientes Episódicos/Não Episódicos

É **episódico** quando a tomada de decisão num determinado instante não depende de episódios anteriores.

É **não episódico** quando há dependência de episódios.

- Cada episódio consiste numa perceção seguida de uma ação.
- O sucesso dessa ação depende apenas do episódio atual.
- Os ambientes episódicos tendem a gerar agentes mais simples, porque estes não precisam pensar no futuro.

## Ambientes Dinâmicos/Estáticos

Se o ambiente mudar enquanto tomo a decisão ele é **dinâmico**, caso contrario é **estático**.

## Ambientes Discretos/Contínuos

Diz-se **discreto** quando origina series de perceções e ações perfeitamente distintas umas das outras. Caso contrario, diz-se **contínuo**.

## Algoritmo Geral de Pesquisa

Uma árvore **Ar** raiz = Estado Inicial, regista os caminhos gerados por aplicação dos operadores do problema aos nós que vão sendo expandidos.

Uma lista **NósPorExpandir** contém os nós fronteira (da árvore) a cada momento. A ordem pela qual os sucessores de um nó são inseridos numa determinada lista, determina qual a variante do AGP que se está a considerar.

Uma lista **NósExpandidos** contém os nós que já foram expandidos. Esta lista evita que o mesmo nó seja expandido várias vezes, **o que poderia gerar loops infinitos** (o mesmo nó pode ser atingidos por vários caminhos).

## Pesquisa em Largura

A partir da raiz, a árvore é **expandida por níveis**. Nós que se encontrem em uma **profundidade N** são expandidos antes dos nós que se encontrem a uma **profundidade N + 1**.

Vantagens:

- **Completa** (procura todas as soluções possíveis e portanto encontrará a ótima, caso exista).
- **Ótima** (desde que o custo do caminho não seja uma função não-decrescente da profundidade de nós – a pesquisa em largura propõe sempre como solução o que tiver menor número de nós. Portanto, se o custo aumentar uniformemente com a profundidade, as soluções com menos nós representam menor custo.

Desvantagens:

- **Elevado custo de pesquisa** – complexidade temporal e exponencial.
- A pesquisa em largura tem uma **complexidade temporal e espacial** de O(b^d) com **b = fator de ramificação** e **d = número de níveis da árvore** (se considerarmos um fator de ramificação de 8, o número de nós expandidos é de 1+8+8^2+8^3+...+8^k
- Problemas de pesquisa cujos algoritmos têm complexidade exponencial apenas podem ser resolvidos para instâncias de pequena dimensão.

## Pesquisa em Profundidade

Cada nó é expandido até ser atingido o **último nível da árvore**, a menos que a solução seja encontrada entretanto.

Características:

- **Incompleta** (no caso da profundidade da árvore ser infinita)
- **Não Ótima** (retorna uma solução qualquer e nenhuma condição pode garantir que seja a melhor)

Vantagens:

- Com **fator de ramificação b** e **profundidade máxima d**, a **complexidade temporal é O(b^d)**, como na pesquisa em largura, porque o número total de nós a gerar é o mesmo.
- A **complexidade espacial** é de apenas O(b.d) porque não há necessidade de ter mais que b.d nós em memória simultaneamente - necessita de pouca memória.

## Pesquisa Uniforme

Variante da pesquisa em largura.

Expandir primeiro os nós que têm um **custo associado menor** (a expansão termina quando for encontrada uma solução e o custo acumulado dos caminhos associados aos nós que falta expandir já for superior à solução encontrada)

Garante a **solução ótima**, bastando apenas que o custo aumente com a profundidade.

- **Completa**
- **Ótima**: desde que o custo aumente com a profundidade g(Sucessor(n)) >= g(n). Se admitirmos que o custo possa diminuir com a profundidade, então seria preciso explorar toda a árvore para determinar o caminho ótimo!

## Pesquisa em Profundidade Limitada

Resolve a limitação da pesquisa em profundidade de não retornar resultados em espaços de profundidade muito grande, **impondo um limite 'm'**, à profundidade máxima a atingir.

**Exemplo:** se um mapa contem 20 cidades, o caminho entre quaisquer duas tem de ser composto, no máximo, por 19. Logo **m = 19**.

- Completa
- Não Ótima
- Complexidade Temporal O(b^m); Complexidade Espacial O(b.m)

## IDS – Pesquisa por Aprofundamento Progressivo

- Combina as pesquisas em largura e profundidade
- Evita a necessidade de se definir 'm' antecipadamente.
- Em vez de se estabelecer um só limite geral, começa por se estabelecer um limite inicial de profundidade = 0.

- Este limite vai-se alargando (1,2,3,...m) para as iterações seguintes (i.e. faz-se uma pesquisa em profundidade de nível 1, depois 2, depois 3,... mas para cada pesquisa reinicia-se o algoritmo da pesquisa em profundidade desde a raiz.
- Ótima, nas condições da pesquisa em largura (custo = função da profundidade)
- Completa
- Complexidade Espacial O(b.m), como a pesquisa em profundidade
- Complexidade Temporal O(b^m), como a pesquisa em profundidade

## Pesquisa Informada

**Motivação:** Pesquisa não informada ineficiente

Métodos do tipo **"best-first"** - o melhor nó é expandido primeiro (função de avaliação - **heurística** - do estado, retorna um valor indicativo da vantagem em expandir esse estado primeiro)

De acordo com a estrutura do AGP, cada nó sucessor é inserido ordenadamente na lista de nós a expandir EM FUNÇÃO DO VALOR DE h(n).

## Pesquisa Sôfrega

- Expande-se em primeiro lugar o nó que parece estar mais perto do objetivo
- Em muitos problemas, pode obter-se uma estimativa do custo do caminho de um dado nó até ao objetivo - **função heurística**. Se **h(n) = 0**, o nó coincide com o **objetivo**. Se >= 0, o nó objetivo pode ser atingido a partir do nó n, sendo o custo estimado h(n). Se h(n) = infinito, o objetivo não pode ser atingido através do nó n.
- Complexidade temporal exponencial - O(b^d), com b = fator de ramificação e d = número de níveis da árvore (máxima profundidade do espaço).
- Complexidade espacial exponencial - O(b^d).
- **Complexidade temporal e espacial** podem ser substancialmente **reduzidas** se h(n) for adequada.
- Não ótima.

- Incompleta (pode seguir caminhos infinitos)

## Pesquisa A*

- Combina a pesquisa uniforme com a sôfrega
- A **Uniforme** "mede" a parte inicial do percurso - g(n)
- A **Sôfrega** "mede" a aparente parte restante - h(n)
- Os custos do caminho provenientes de ambas podem combinar-se numa simples soma **f(n) = g(n) + h(n)**: "mede" o custo estimado da solução que passa pelo nó n. No AGP, a inserção em **NosAExpandir** é feita por ordem crescente de f(n).
- A pesquisa A* é ótima e completa desde que:
    - A **HEURÍSTICA UTILIZADA** nunca sobestime o custo do caminho do nó n até ao objetivo (isto é, nunca possa assumir um valor superior ao do custo real) - **HEURÍSTICA ADMISSIVEL.**

## Variantes A* - Limitação de memória

IDA*: A* com aprofundamento progressivo "IDS para A*"

Está para a pesquisa A* como IDS está para a pesquisa em profundidade.

- No IDS, cada iteração é limitada por um nível de profundidade crescente
- No IDA* cada iteração é limitada por um valor crescente da função de custo, f(n) = g(n) + h(n)
- Para cada "limite de custo estimado", fi, "exclui" os nós cujo valor f é superior
- Pára quando atingir um nó objetivo cujo f é <= que o limite atual
- Enquanto não encontrar um objetivo nestas condições, progride para o limite seguinte, fi+1, que pode provir de outro nó situado à mesma profundidade do que proporcionou o limite anterior, fi. O IDA* é controlado pelo valor de f e não pela profundidade d do nó. É

determinado na iteração i, escolhendo o menor custo estimado de entre todos os custos estimados associados aos nós por expandir

- Completa e ótima
- Por ser baseada na pesquisa em profundidade: o requerimento de memória é baixo e pode ser aproximado por b.d (b branching factor, d profundidade da solução)

**SMA*: Simplified Memory Bounded A***, desenhado para não ultrapassar o limite de memória disponível para resolver um problema

- Completo e ótimo desde que a memória possibilite a sua execução completa
- Se a memória estiver toda utilizada devido às expansões efetuadas, "esquece" os nós menos promissores (os de valor de f mais elevado), usando o espaço assim libertado para o resultado de outras expansões
- O nó a expandir é o de menor valor de f, porém, quando se expande esse nó, adiciona-se-lhe apenas um sucessor por cada iteração
- Quando um nó se encontrar completamente expandido, o seu custo estiamdo, f, é atualizado com o mínimo dos valores de f dos seus nós filhos da iteração

## Problemas com Restrições

Trata-se de um problema cuja solução só é válida se **satisfizer certas condições**:

- **Variáveis:** os seus valores finais representação a solução
- **Domínio:** Conjunto de valores que as variáveis podem assumir
- **Restrições:** atuam sobre as variáveis
- **Problema:** Assinar valores às variáveis sem violar as restrições
- Interessa determinar um "estado" final válido e não um caminho que leve a esse estado. O estado final é desconhecido e constitui a solução do problema.

- Exemplo: problema das 8 rainhas
- Um CSP pode ser resolvido por técnicas de pesquisa, contudo são geralmente ineficientes neste contexto, dado gerarem muitos estados desnecessariamente
- Algoritmos especialmente adaptados à resolução de CSPs: Hill-Climbing, Simulated-Annealing, Pesquisa Tabu

## Melhoramento Iterativo

Não anotam estados intermédios que conduzem a uma solução, apresentando apenas a configuração válida que a compõe

Partem de uma configuração inicial completa (que viola as restrições), eventualmente gerada aleatoriamente, e melhoram-na sucessivamente até alcançarem uma solução

## Trepa-Colinas

Parte de um estado inicial dado ou gerado aleatoriamente. Todas as variáveis com valores atribuídos.

Gera os estados sucessores do estado atual (**VIZINHOS**).

Através de uma função de avaliação, avalia cada estado assim gerado e escolhe o de maior valor.

Para quando o estado selecionado tiver um valor inferior ao escolhido na iteração anterior (significa que a solução "piorou" e que se está a "descer a colina" em vez de a "subir".

**Problemas:** um máximo local pode ser atingido sem que corresponda ao máximo absoluto (melhor solução).

Nos "planaltos" é necessário escolher uma direção aleatoriamente.

Um cume pode ter lados tão inclinados que o passo seguinte conduz ao "outro lado do cume" e não ao seu topo. Neste caso a solução poderá "oscilar" nunca atingindo o máximo pretendido.

**Tentativa de resolução dos problemas relativos a atingir um ponto de não progresso:** Reiniciar a pesquisa partindo de um estado inicial diferente (Random-Restart-Hill-Climbing).

Guarda o melhor resultado obtido nas pesquisas anteriores (até ao ponto de não-progresso).

Para quando atingir o número de reinícios máximo ou quando o melhor resultado guardado não for ultrapassado durante 'n' iterações (valor de 'n' é pré-fixado).

Variantes:

- Permitir o deslocamento ao longo de um planalto;
- First-Choice: Visita vizinhos de forma aleatória, aceita um vizinho de melhor qualidade e termina iteração (útil quando a vizinhança é grande, algoritmo não determinista)
- Random Restart (diversos pontos de partida)

## Simulated Annealing

Quando encontra um máximo (pode ser apenas um local) o algoritmo prossegue "durante algum tempo" a pesquisa no sentido descendente.

Em vez de escolher sempre o estado seguinte de maior valor, escolhe-se um, aleatoriamente.

Se a sua avaliação for superior à do estado anterior, **É SEMPRE ESCOLHIDO.**

Se for inferior, é escolhido mas apenas com uma certa probabilidade (<1) que baixa à medida que um parâmetro 'T' tende para zero ao longo das sucessivas iterações

Quando T for muito pequeno, a escolha de estados de pior avaliação quase nunca ocorre, e o "Simulated Annealing" comporta-se (quase) como o "Hill-Climbing".

**Probabilístico:** resultado não determinista, deve-se executar o algoritmo mais do que uma vez.

Se o arrefecimento for "suficientemente" lento é sempre atingido o ótimo global.

## Pesquisa Tabu

Durante a pesquisa, forçar a exploração de novas zonas do espaço de procura (pode assim evitar-se entrar em ciclos).

Implementação: recurso a uma memória de curta-duração (indica quais os movimentos proibidos - tabu).

Vantagens:

- Escolhe sempre o melhor vizinho, desde que seja válido, exibindo assim um comportamento determinista
- Ao aceitar soluções de pior qualidade, pode evitar ótimos locais

Desvantagens:

- Nem sempre é fácil ajustar o limite de memória e número máximo de iterações

## Algoritmos Genéticos

Sub-classe da computação evolucionária, baseados na teoria da evolução de Darwin.

Funcionamento:

- **Seleção:** As "melhores hipóteses" são as de maior "aptidão". Esta aptidão é avaliada por uma função.
- **Recombinação (crossover) e Mutação:** Em vez de procurarem sistematicamente uma solução (hipótese h), os AGs geram hipóteses sucessoras das atuais (offspring) recombinando probabilisticamente as melhores hipóteses entre si, e "mutando" algumas outras.

- **Seleção proporcional:** A probabilidade da seleção de uma hipótese é proporcional ao **quociente q** entre a sua aptidão e a soma das aptidões restantes (as hipóteses de maior valor de q são selecionadas mais vezes)

## Método da roleta

Cada hipótese de uma dada população possui uma fitness fi.

f1 = 1/6, f2 = 1/3, f3 = 1, f4 = 1/2

Calculam-se os valores acumulados:

A = f1 = 1/6

B = f1 + f2 = 1/2

C = f1 + f2 + f3 = 3/2

D = f1 + f2 + f3 + f4 = 2

Normalizam-se estes valores:

A = 0.0833

B = 0.25

C = 0.75

D = 1

Gera-se um número aleatório x entre 0 e 1 e verifica-se sobre qual das hipóteses ele "cai"

Como qualquer x é igualmente provável, ele cairá mais vezes sobre a zona correspondente à hipótese que ocupa maior espaço na reta.

**Seleção por torneio:**

Selecionar k hipóteses (tsize) de entre a pop. De entre elas, selecionar a de maior fitness. Duas hipóteses são selecionadas aleatoriamente de entre a população. Com uma probabilidade pré definida p, a de maior aptidão é selecionada (a outra é selecionada com probabilidade (1-p)).

**Seleção por Posicionamento (Ranking Selection):**

As hipóteses são ordenadas de acordo com a sua aptidão, da melhor para a pior. O valor do ranking (posição depois da ordenação) é usado (em vez da aptidão) por uma função que determina a probabilidade de seleção da hipótese (o espaço que ocupará na roleta).

**Recombinação:**

As hipóteses são, muitas vezes, representadas por strings, o que permite uma implementação simples das operações de recombinação e mutação.

## Algoritmos para Jogos

Diferenciam-se pela inclusão de um fator de incerteza devido à presença de um adversário.

Incerteza do tipo **não probabilística**: o adversário B tentará a melhor jogada para ele, o que implica a pior jogada para o oponente A. A aplicação de algoritmos de pesquisa para encontrar a melhor solução para A não funciona, pois é necessário contar com os movimentos de B.

## MiniMax

- Jogos **determinísticos** e **observáveis**
- Jogo com **dois indivíduos**: MAX e MIN
- Jogam alternadamente: **MAX joga primeiro**
- No final do jogo, MAX ganha / MIN ganha / empate (pode ser guardado o score ou 1, -1, 0)
- Seleção da melhor jogada por parte de cada jogador
- **Estado inicial:** posição inicial, valor das "peças" e indicação de quem inicia o jogo
- Operadores

- Teste de Final
- **Função de Utilidade:** mede o "proveito" que o estado terminal alcançado representa para cada um dos jogadores
- MAX deve conhecer previamente os valores de todos os estados terminais
- Partir do princípio que MIN jogará de forma a prejudicar MAX

1. Gerar a árvore do jogo
2. Determinar a Utilidade de cada estado terminal (valor para MAX)
3. Progredir para o nível anterior (neste nível é MIN que joga) - A cada nó assinalar o valor mínimo dos nós seus filhos (isto traduz que MAX espera que MIN jogue de modo a minimizar a pontuação de MAX)
4. Progredir para o nível anterior (neste nível é MAX que joga): A cada nó assinalar o valor máximo dos nós seus filhos (isto traduz que MAX jogará da melhor forma)
5. Prosseguir assim até ser atingida a raiz da árvore.

Toda a árvore é percorrida, em **profundidade**.
**Algoritmo recursivo:** atribuição de valores é feita dos nós terminais para a raiz
Impraticável para jogos complexos.

## Alpha-Beta Pruning

Requer consideravelmente menos recursos de memória e tempo, mesmo para jogos relativamente simples.

O algoritmo baseia-se na utilização de dois parâmetros, “Alpha” e “Beta”:

- **Alpha:** representa o valor mínimo garantido que **MAX** poderá obter.

Como representa um limite inferior é **inicializado a -inf** e vai crescendo, sendo atualizado num nó MAX.

**Beta:** representa o valor máximo que **MIN** consegue impor a MAX

- MAX nunca conseguirá jogar para obter um valor superior a beta

- Sendo um limite superior, é inicializado a +inf e posteriormente vai decrescendo (atualizado num nó MIN).
- Se ALPHA (melhor hipótese para MAX até então) >= BETA (melhor hipótese para MIN até então), **corta-se o ramo**.
- Algoritmo ótimo.
- Eficácia depende da ordem pela qual os sucessores são avaliados.

- Nó filho herda Alfa e Beta do pai.
- Se esse nó filho for um MIN, atualiza apenas o Beta com o valor da pior opção para MAX (sendo Beta aqui menor que Alfa, cortar restantes filhos | sendo beta aqui maior, ver restantes ramos | se depois de ver todos os ramos continuar maior, atualizar Alfa no nó pai).
- Se esse nó filho for um MAX, atualiza apenas o Alfa com o valor da melhor opção para MAX (tem de ver todos os nós pois não sabe qual a melhor).

## Jogo Com Elemento Sorte

- Calcula-se a utilidade nos estados terminais
- Nos nós superiores **Max** obtém-se o maior valor (como no **MiniMax**)
- Nos nós superiores **Min** obtém-se o menor valor (como no **MiniMax**)
- Nos **nós sorte**, calcular o valor esperado:

para **MAX**:

**E = Somatório de i = 1 até n filhos** => P(di)*max(utilidade(s))

para **MIN**:

**E = Somatório de i = 1 até n filhos** => P(di)*min(utilidade(s))

# Redes Neuronais

**Aprendizagem automática:** modificação das sinapses existentes; criação de novas ligações.

**Mecanismo de aprendizagem:** supervisionada / por reforço / não supervisionada.

**Rede Neuronal Artificial:**

- Elevado número de interconexões entre unidades de processamento elementares.
- O conhecimento é armazenado através dos valores dos pesos, obtidos através de um processo de adaptação ou aprendizagem a partir de um conjunto de dados de treino.
- O ajusto dos pesos – **Aprendizagem** é realizada de forma automática.
- Caracteriza-se por um processamento distribuído.

**Aprendizagem:** Processo pela qual os parâmetros de uma rede neuronal são adaptados através de um processo de treino baseado em dados experimentais; O tipo de aprendizagem determina a forma de adaptação dos parâmetros.