Pós-Graduação Lato Sensu
Curso de Especialização em Inteligência Artificial

Introdução a Inteligência Artificial

Prof. Dr. **Lucas Dias Hiera Sampaio**

PARTE 03

UTFPR
UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CAMPUS CORNÉLIO PROCÓPIO

Problemas
e
Busca

1. Apresentação

Na aula desta semana vamos estudar os fundamentos do processo de resolução de problemas e como algoritmos de busca podem ser aplicados neste contexto. Para isso iremos discutir desde a fundamentação matemática dos problemas relacionados à IA e exemplos de algoritmos simples que podemos utilizar para solucioná-los.

2. Agentes e Problemas

No contexto de Inteligência Artificial agente é tudo o que pode ser considerado capaz de perceber seu ambiente por meio de sensores e agir sobre esse ambiente por meio de atuadores. Por exemplo, um robô de limpeza pode possuir sensores em sua base para verificar se o chão está sujo e reagir de acordo: caso esteja sujo, executa a limpeza e posteriormente se movimenta ou, caso não esteja sujo, apenas se movimenta. A Figura 1 apresenta um exemplo esquemático de agente que atua de forma reativa.

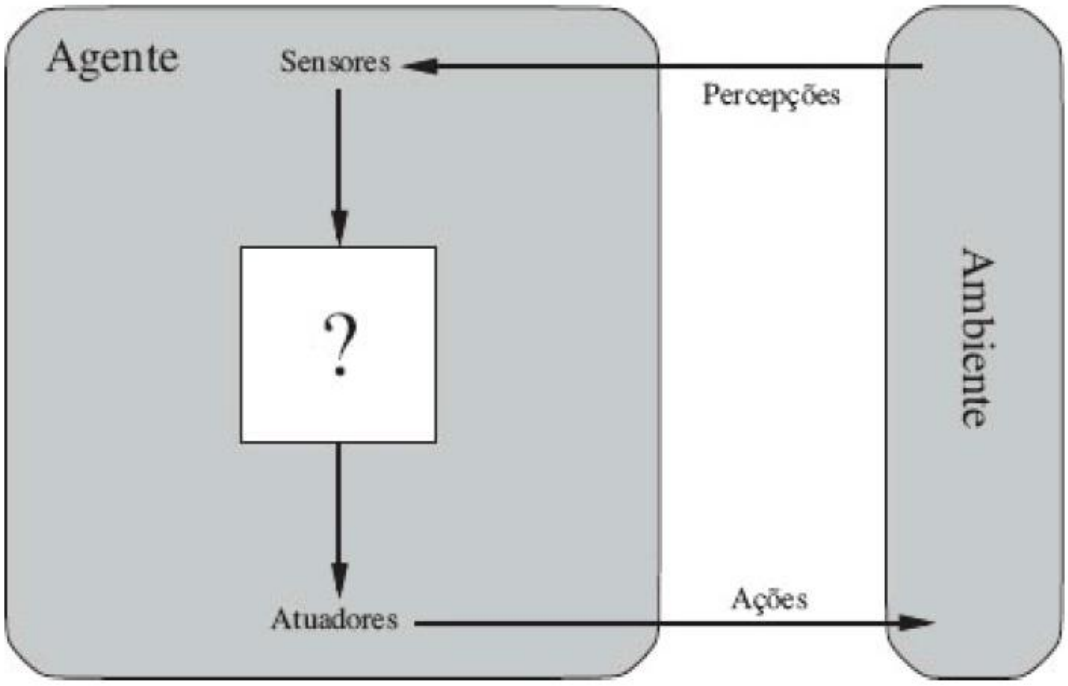

Figura 1: Exemplo de agente reativo interagindo com o ambiente. Fonte: Russel e Norvig (2013).

No exemplo acima, se o agente possuísse muitas funções, ou o próprio ambiente fosse complexo demais para navegar de forma simples teríamos uma estrutura com muitas condições, situações e possibilidades, tornando impraticável o uso de um agente reativo como o exemplificado acima. Nestas situações, o uso de um agente de resolução de problemas é fundamental para o sucesso da execução de tarefas.

Antes de compreender a fundo como funcionam e como podem ser criados os agentes de resolução de problemas, é fundamental compreender conceitos como **objetivo**, **domínio do problema**, **soluções**, **solução ótima**, entre outros.

Denominamos **objetivo** do problema o estado ou resposta que o agente busca alcançar por meio de suas interações com o ambiente e/ou domínio do problema. O **domínio do problema**, por sua vez, é o conjunto de todas as possíveis soluções para o problema levando em consideração as **restrições** que podem existir. Uma **solução** do problema é um elemento do domínio do problema que satisfaz as restrições e uma **solução ótima** é aquela que dentre todas as soluções possíveis apresenta a melhor avaliação quando mensurada por uma **função objetivo**, isto é, por uma métrica capaz de aferir quão boa a solução é. As soluções **ótimas** possuem os melhores valores da métrica que avalia se o objetivo foi alcançado.

Para que cada um desses conceitos fique mais claro, iremos utilizar na sequência alguns exemplos de problemas que podem ser solucionados utilizando técnicas de inteligência artificial.

Exemplo 1: suponha que um agente esteja na praia de Copacabana no Rio de Janeiro aproveitando uma viagem de férias. O objetivo do agente pode envolver diferentes fatores como: melhorar o bronzeado, observar paisagens, apreciar a vida noturna, evitar ressacas, etc. A tomada de decisão neste caso é complexa e envolve participar de muitas atividades e ler guias de viagens, uma vez que não há restrições no problema, apenas muitos objetivos a serem cumpridos.

Exemplo 2: Suponha um agente na mesma situação do anterior, porém são oito horas da manhã e o agente deverá tomar um voo de volta para sua residência às dez horas da manhã do dia seguinte. Neste novo exemplo, temos uma restrição: o tempo. Logo, faz sentido para esse agente traçar como objetivo realizar a maior quantidade de atividades possíveis no tempo que lhe resta. Os objetivos podem ter pesos diferentes, por exemplo, o agente pode achar muito mais prazeroso permanecer na praia tomando banho de sol do que observar paisagens, ou até mesmo serem conflitantes: apreciar a vida noturna tende a envolver ressacas no dia seguinte.

Exemplo 3: Suponha um novo agente diferente dos exemplos anteriores. Neste caso, o agente é um carro autônomo utilizado por serviços de entrega que precisa se deslocar entre duas cidades para completar uma entrega. O agente deve sair da cidade 1 e se deslocar até a cidade 6 para completar seu objetivo (Veja Figura 2).

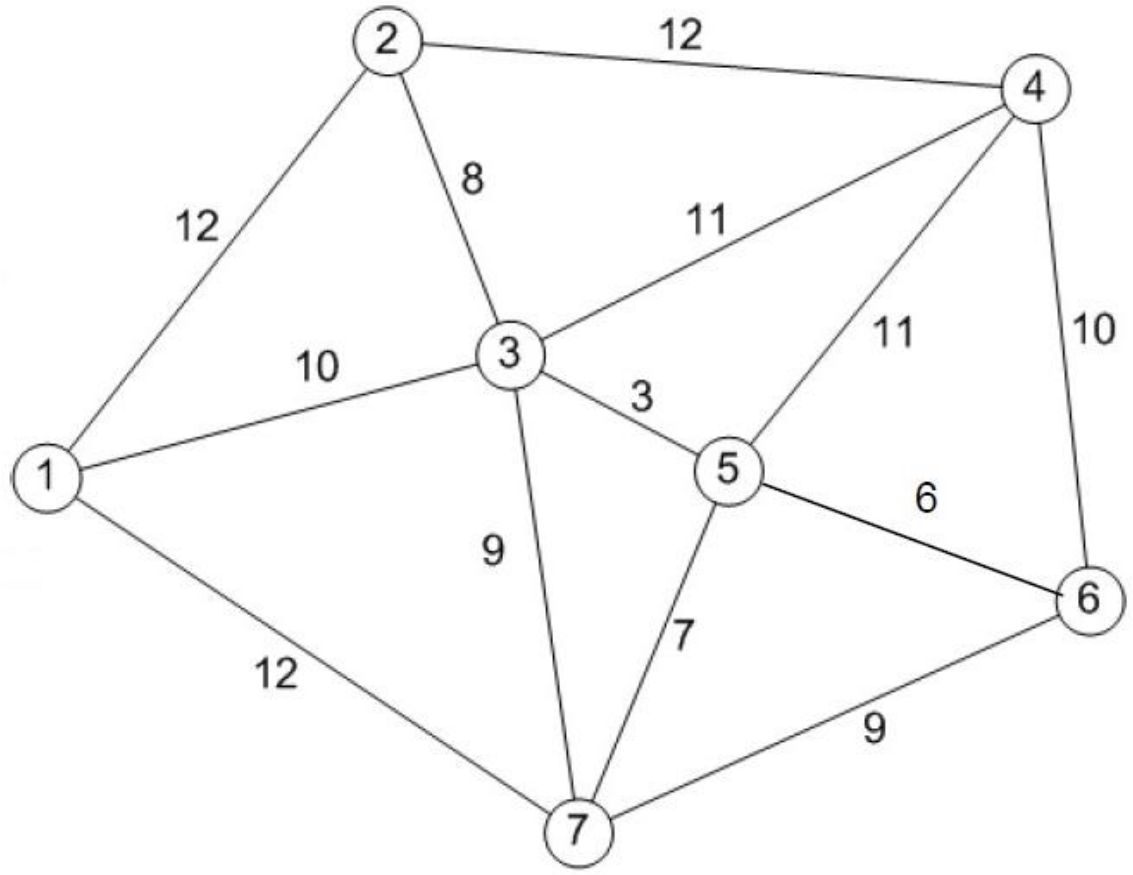

Figura 2: Mapa de cidades (círculos) e distâncias (segmentos de reta) do exemplo 3.

Encontrar uma solução para o problema do exemplo 3 é um desafio grande: trata-se de encontrar o menor caminho possível em um grafo. Uma das formas mais simples de solucionar o problema é por meio da técnica de **gerar e testar**. Neste caso poderíamos criar um caminho aleatório entre as cidades 1 e 6 (**gerar**) e mensurar a distância deste caminho (**testar**), a solução do problema seria o caminho gerado com a menor distância. Esta técnica também é conhecida como **busca exaustiva** ou **busca por força bruta**.

Considerando um sistema de geração de soluções aleatórias, para encontrar o melhor caminho entre as cidades 1 e 6 precisaríamos testar todos os possíveis caminhos entre as cidades. Se não houver restrições e o grafo possuir ciclos (como o da Figura 2) o número de possíveis caminhos é infinito e o tempo necessário para encontrar a resposta torna a resolução do problema infactível por esta técnica.

A fim de permitir que agentes sejam capazes de solucionar problemas como este, diferentes **algoritmos de busca** (busca da solução do problema) foram propostos na literatura e vamos investigar os principais na seção seguinte.

3. Algoritmos de Busca

Os algoritmos discutidos nesta seção podem ser utilizados para solucionar problemas como o postulado no **Exemplo 3** da seção anterior, ou seja, problemas cujo domínio é **discreto** (formado por um conjunto contável).

Nos algoritmos de busca em domínio discreto, cada elemento da solução é chamado de **estado não-objetivo** e o elemento final (cidade de destino no **Exemplo 3**) é chamado de **estado objetivo**. Sendo assim, algoritmos de busca que são capazes de avaliar qual estado não-objetivo é mais promissor para gerar uma solução melhor são denominados **algoritmos de busca informada ou busca heurística**. Já os algoritmos de busca que não avaliam o estado não-objetivo são denominados **busca sem informação** ou **busca cega**.

As **heurísticas** serão objeto de estudo de disciplina no Módulo 2 do curso de Especialização e portanto não serão discutidas aqui.

Entre os diferentes algoritmos de busca cega capazes de gerar diferentes caminhos entre duas cidades conforme postulado na seção anterior, a **busca em largura** (do inglês, *Breadth-First Search*, BFS) é um algoritmo simples no qual o grafo (como da Figura 2) é utilizado para construir uma árvore de busca. Partindo do nó inicial, cada nível da árvore mapeia os possíveis próximos nós para formar uma rota entre a cidade inicial e a cidade final.

De forma geral, a busca em largura pode ser feita conforme o código **python** abaixo:

```
visitados = [] #Lista de nós visitados.
fila = []      #Inicialização da fila

def bfs(visitados, grafo, no): #algoritmo BFS
  visitados.append(no)
  fila.append(no)

  while fila:           # Loop para visitar todos os nós
    m = fila.pop(0)

    for vizinho in grafo[m]:
      if vizinho not in visitados:
        visitados.append(vizinho)
        fila.append(vizinho)
```

Para uma visualização do algoritmo acima, assista ao vídeo disponível no Moodle “Animated Visualization BFS Algorithm”.

Quando a distância entre as cidades for igual, o algoritmo BFS vai sempre encontrar a menor distância entre a cidade inicial e a cidade final. Todavia, quando estas

distâncias forem diferentes umas das outras é necessário utilizar outro algoritmo de busca que leve em consideração os valores da distância. É o caso da **busca de custo uniforme** onde o nó do grafo a ser expandido é sempre o com o menor custo de caminho, isto é, a busca continua sempre no sentido do nó cujo caminho entre a cidade inicial e ele seja sempre o menor.

O código abaixo em **python** apresenta uma implementação da busca de custo uniforme.

```
def  busca_custo_uniforme(destino, inicio):

    global grafo, custo
    resultado = []

    # Fila de Prioridades
    fila = []

    # Inicia o vetor resultado
    for i in range(len(destino)):
        resultado.append(10**8)

    # Insere na fila o primeiro elemento
    fila.append([0, inicio])

    # Mapeia os nós visitados
    visitados = {}

    # Contador
    contador = 0

    # Enquanto a fila não estiver vazia
    while (len(fila) > 0):

        # Ordena a fila (crescente)
        fila = sorted(fila)
        # Usa o último elemento (menor)
        p = fila[-1]

        # Remove o menor elemento da fila
        del fila[-1]

        # Obtém o valor original da distância
        p[0] *= -1

        # Verifica se o nó é o destino
        if (p[1] in destino):

            # Recupera o indexador
            index = destino.index(p[1])

            # Se um novo destino foi atingido
            if (resultado[index] == 10**8):
                contador += 1

            # Se o custo for inferior, atualiza o custo
```

```
                if (resultado[index] > p[0]):
                    resultado[index] = p[0]

                # Remove o nó da fila
                del fila[-1]

                fila = sorted(fila)
                if (contador == len(destino)):
                    return resultado

            # Verifica os nós não visitados que são adjacentes ao nó
            # atual
            if (p[1] not in visitados):
                for i in range(len(grafo[p[1]])):

                    # o valor é multiplicado por -1 de tal forma que
                    # a menor prioridade está no topo
                  fila.append( [(p[0] + custo[(p[1], graph[p[1]][i])])*
-1, grafo[p[1]][i]])

            # Marca como visitado o nó atual
            visitados[p[1]] = 1

        return resultado
```

É importante notar a diferença entre os dois algoritmos: embora possam ser utilizados no mesmo problema, o BFS tem como principal objetivo encontrar as diferentes rotas existentes entre o nó de origem e o nó de destino enquanto o algoritmo de busca uniforme tem como objetivo encontrar o menor caminho entre a origem e o destino.

Na próxima seção vamos discutir os fundamentos de complexidade computacional que muitas vezes limitam os avanços que podem ser feitos na área uma vez que alguns algoritmos podem necessitar de muito tempo e memória para encontrar alguma solução.

4. Complexidade Computacional

Na computação, o desenvolvimento de novos algoritmos é geralmente acompanhado de uma análise de complexidade uma vez que os recursos são finitos. É importante frisar que muitos problemas clássicos são extremamente difíceis de solucionar e essa dificuldade tende a crescer conforme o domínio do problema aumenta, isto é, solucionar o problema da Figura 2 é simples comparado a um problema equivalente porém com 1 milhão de nós.

A complexidade computacional de algoritmos pode ser calculada em duas dimensões: tempo, que geralmente está diretamente relacionado a velocidade de operação do processador (número de operações por segundo), e quantidade de memória.

Em termos de tempo, de forma geral, utiliza-se a notação **big-O**: $O(n)$ onde n é o tamanho da entrada do algoritmo por exemplo, significa que o tempo de execução cresce linearmente com a entrada. Já $O(n^2)$ indica que a complexidade é quadrática, isto é, o tempo de execução cresce conforme uma equação do segundo grau com relação ao tamanho da entrada. A complexidade $O(2^n)$ é chamada de exponencial e o tempo de execução cresce exponencialmente em relação ao tamanho da entrada. Por fim, $O(n!)$ é a complexidade fatorial.

Além destas complexidades, existem também a complexidade constante $O(c)$, a logarítmica $O(log(n))$ e a log-linear $O(n\ log(n))$. A Figura 3 ilustra o crescimento do tempo de execução conforme o tamanho da entrada para cada uma das complexidades listadas. Note que a complexidade fatorial pode ser considerada a pior (tem a maior taxa de crescimento) enquanto a constante é independente do tamanho da entrada.

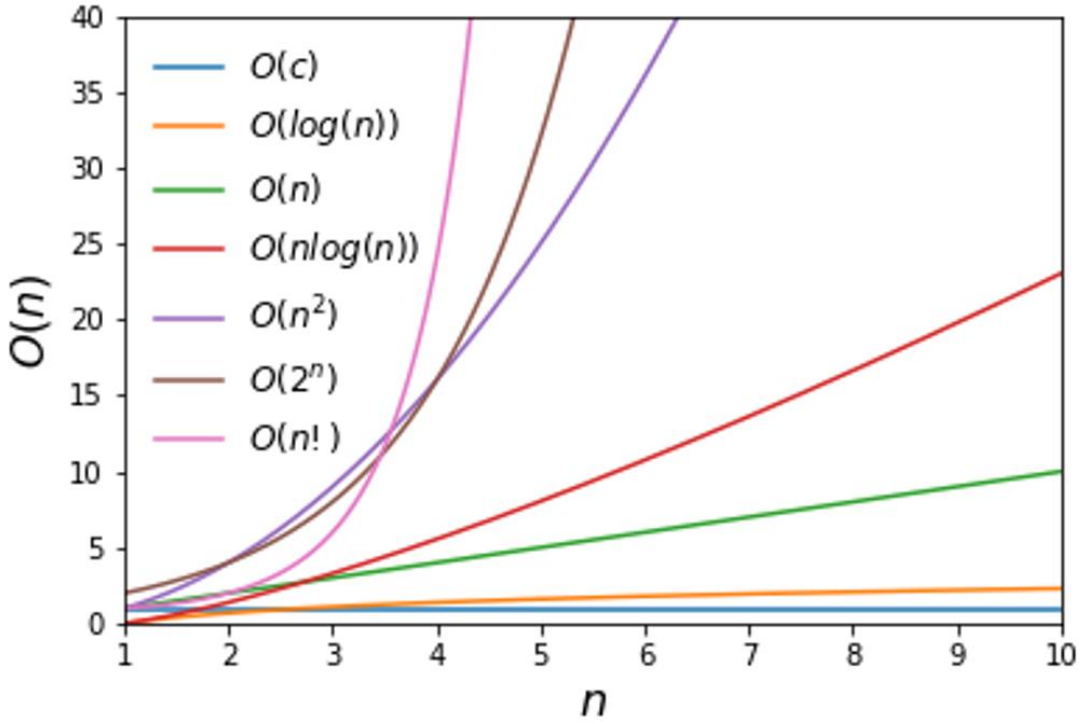

Figura 3: Crescimento do tempo de execução para diferentes complexidades computacionais.

A complexidade em termos de memória também é um fator determinante uma vez que a memória computacional não é um recurso infinito. Para ilustrar os desafios que a complexidade impõe a execução de algoritmos suponha o algoritmo BFS será utilizado para criar a árvore de todos os possíveis caminhos de um grafo. Suponha que cada nó esteja conectado a 10 outros nós no grafo, que cada nó necessita de 1000

bytes de memória e o computador seja capaz de processar um milhão de nós por segundo. A Tabela 1 apresenta a quantidade de tempo e de memória necessária para concluir uma única execução do BFS para diferentes tamanhos de grafo de entrada.

Tabela 1: Tempo necessário e memória para execução do BFS com diferentes grafos de entrada assumindo 1KB por nó e processamento de 1 milhão de nós por segundo.

Profundidade da Árvore	**Nós**	**Tempo**	**Memória**
2	110	0,11 ms	107 KB
4	11110	11 ms	10,6 MB
6	1 milhão	1.1 s	1 GB
8	100 milhões	2 minutos	103 GB
10	10 bilhões	3 horas	10 TB
12	1 trilhão	13 dias	1 PB
14	100 trilhões	3,5 anos	99 PB
16	10 quatrilhões	350 anos	10 EB

Note que para a busca em largura (BFS) a memória é um problema maior que o tempo de execução - veja que embora 10 bilhões de nós leve apenas 3 horas para concluir ele necessita de 10 Terabytes de memória. Já 10 quatrilhões de nós precisará de 10 exabytes de memória (este número é 1 seguido de 19 zeros, ou 10 000 000 000 000 000 000 bytes).

Nas próximas aulas continuaremos a aprender outros algoritmos de busca e vamos ver o seu comportamento quando executando no computador. Para isso vamos configurar um ambiente de desenvolvimento e comparar os diferentes tempos de execução.