Roteiro de Aula 2 – Busca (Cap. 3.1–3.5, Poole & Mackworth, AIFCA 3e)

Disciplina: BCC740 – Inteligência Artificial

7 de outubro de 2025

Sumário da Aula

1. Introdução: Agentes e Ação Inteligente
2. 3.1 **Problem Solving as Search**
3. 3.2 **State Spaces**
4. 3.3 **Graph Searching**
5. 3.4 **A Generic Searching Algorithm**
6. 3.5 **Uninformed Search Strategies**

1 Agentes e Ação Inteligente (Contexto do Capítulo 3)

Definição de Agente e Agente Inteligente

Um **agente** é algo que **age em um ambiente**: ele percebe (recebe informações do ambiente) e executa ações no mundo. (*Poole & Mackworth, 3e; fontes:* `https://www.inf.unibz.it`, `https://artint.info`)

Um **agente inteligente** é aquele agente que **age de forma inteligente** — ou seja, cujas ações não são arbitrárias, mas adaptadas às suas metas, à situação e às limitações. (*Fontes:* `https://artint.info`, `https://www.cs.ubc.ca`)

Características de um agente que age inteligentemente

De acordo com Poole & Mackworth, um agente age inteligentemente quando:

(i) Suas ações são apropriadas às suas metas e às circunstâncias do ambiente.
(ii) Ele é flexível frente a mudanças no ambiente e nas metas.
(iii) Ele aprende com a experiência — melhora seu desempenho ao longo do tempo.
(iv) Ele faz escolhas apropriadas levando em conta suas limitações perceptuais e computacionais (memória, tempo, custo).

Síntese conceitual

Em outras palavras, a **inteligência de um agente** é medida pelo quão bem ele consegue **agir bem**, dadas:

- suas percepções (o que pode observar),
- suas capacidades (o que pode calcular e executar),
- suas metas (o que busca alcançar).

Relação com o tema da aula: Resolver problemas como busca é uma das formas mais fundamentais de ação inteligente. Ao formalizar o raciocínio como uma sequência de decisões que leva de um estado inicial a um estado objetivo, estudamos **como um agente pode planejar suas ações racionalmente**.

2 Problem Solving as Search

Ideia central

Resolver um problema como **busca** em um espaço de estados: encontrar uma sequência de ações que transforma o estado inicial em um estado meta, minimizando (opcionalmente) um custo.

Formulação

Um problema de busca é uma quíntupla

$$\langle S, A, \gamma, c, (s_0, G) \rangle$$

onde:

- S: conjunto (possivelmente grande) de estados.
- $A(s)$: conjunto de ações aplicáveis em $s \in S$.
- $\gamma(s, a)$: função de transição (ou $T(s, a) \to s'$).
- $c(s, a, s') \geq 0$: custo do passo; custo de caminho $g(n)$ é a soma acumulada.

- $s_0 \in S$: estado inicial; $G \subseteq S$: conjunto de metas (teste de objetivo).

Solução Uma **solução** é uma sequência de ações $(a_1, \ldots, a_k)$ tal que $\gamma(\ldots\gamma(\gamma(s_0, a_1), a_2)\ldots, a_k) \in G$. Quando há custo, buscamos solução de **custo mínimo**.

3 State Spaces

Representação de estados e ações

- **Estados**: escolhas de representação têm impacto em eficiência (ex.: tuplas imutáveis para puzzles).
- **Ações**: operadores locais; podem depender do estado (pré-condições).

4 A Generic Searching Algorithm

Nós de busca

Cada nó mantém: estado, pai, ação geradora, profundidade, custo $g(n)$.

```
from collections import deque
import heapq

class Node:
    __slots__ = ("state","parent","action","depth","g")
    def __init__(self, state, parent=None, action=None, g=0):
        self.state  = state
        self.parent = parent
        self.action = action
        self.g      = g
        self.depth  = 0 if parent is None else parent.depth + 1

def solution(n):
    """Reconstroi a sequencia de acoes/estados a partir do no meta."""
    path = []
    while n and n.parent is not None:
        path.append(n.action)
        n = n.parent
    return list(reversed(path))
```

Algoritmo Genérico de Busca em Grafo

```
def generic_graph_search(s0, is_goal, successors, add_to_frontier, pop_frontier):
    start = Node(s0)
    if is_goal(s0):  # teste de objetivo no no inicial
        return []
    frontier = add_to_frontier(None, start)  # inicializa fronteira
    explored = set()                          # estados ja expandidos

    while frontier:
        node, frontier = pop_frontier(frontier)  # escolhe politica de expansao
        if node.state in explored:
            continue
        explored.add(node.state)

        for (a, s_next, cost) in successors(node.state):
            child = Node(s_next, parent=node, action=a, g=node.g + cost)
            if is_goal(s_next):
                return solution(child)
            if s_next not in explored:
                frontier = add_to_frontier(frontier, child)
    return None  # falha
```

Observação A **política de fronteira** define a estratégia (Seção 3.5). Para busca de custo uniforme, por exemplo, a fronteira é uma `priority queue` por $g(n)$; para BFS, uma `queue`; para DFS, uma `stack`.

5 Exemplo: O Problema do 8-Puzzle

O **8-puzzle** é um dos exemplos mais clássicos de *problema de busca em espaço de estados*. Ele ilustra como representar estados, operadores e funções de custo, e serve de base para o estudo posterior de buscas informadas (heurísticas, Seção 3.6).

Descrição informal

O tabuleiro consiste em uma grade 3×3 contendo oito peças numeradas de 1 a 8 e uma casa vazia. O objetivo é mover as peças deslizantes, uma por vez, até atingir uma configuração final desejada (estado meta).

Inicial

2	8	3
1	6	4
7		5

$\Rightarrow$

Meta

1	2	3
8		4
7	6	5

A casa vazia (representada por "_") pode ser movida para cima, baixo, esquerda ou direita, trocando de lugar com a peça adjacente.

Representação de estados

Um estado é uma permutação dos números $\{1, \ldots, 8\}$ e do espaço vazio "_", representada, por exemplo, como uma tupla de nove elementos:

$$S = \{(x_1, x_2, \ldots, x_9) \mid x_i \in \{1, \ldots, 8, \_\}\}.$$

Estado inicial e meta

$$s_0 = (2, 8, 3,\ 1, 6, 4,\ 7, \_, 5), \qquad G = \{(1, 2, 3,\ 8, \_, 4,\ 7, 6, 5)\}.$$

Ações possíveis

O espaço vazio pode se mover nas quatro direções, quando possível:

$$A = \{\text{mover\_cima},\ \text{mover\_baixo},\ \text{mover\_esquerda},\ \text{mover\_direita}\}.$$

As ações dependem da posição atual do espaço vazio.

Função de transição

Seja $\gamma(s, a)$ a função de transição que aplica a ação a ao estado s, trocando o espaço vazio com a peça adjacente correspondente.

```python
def successors(state):
    """Retorna todos os estados sucessores do 8-puzzle."""
    idx = state.index("_")
    moves = []
    row, col = divmod(idx, 3)
    directions = {
        "up":    (-1, 0),
        "down":  ( 1, 0),
        "left":  ( 0,-1),
        "right": ( 0, 1)
    }
    for act, (dr, dc) in directions.items():
        nr, nc = row + dr, col + dc
        if 0 <= nr < 3 and 0 <= nc < 3:
            nidx = 3*nr + nc
            new_state = list(state)
            new_state[idx], new_state[nidx] = new_state[nidx], new_state[idx]
            moves.append((act, tuple(new_state), 1))
    return moves
```

Custo e objetivo

- Custo de cada passo: $c(s, a, s') = 1$ (cada movimento tem custo unitário).
- Teste de objetivo: $s = G$.

Espaço de busca

O número total de possíveis arranjos é $9! = 362{,}880$, mas apenas metade desses (181,440) são **atingíveis** a partir de qualquer estado dado — o espaço de estados é particionado em dois conjuntos disjuntos (paridade de permutação).

Propriedades importantes

- O fator de ramificação médio é $\bar{b} \approx 2.13$ (nem todas as quatro direções são sempre possíveis).
- A profundidade da solução ótima típica varia de 10 a 30 movimentos.
- O problema é ideal para demonstrar buscas **em largura (BFS)** e **de custo uniforme (UCS)**.

Observações

- O 8-puzzle é um caso particular do **n-puzzle**, cuja versão 15-puzzle (4×4) é notoriamente mais difícil.
- Ele fornece uma boa base para o estudo de **buscas informadas** (heurísticas como distância de Manhattan ou número de peças fora do lugar).
- Em busca cega, a **BFS** encontra a solução mínima, enquanto a **DFS** pode gerar ciclos e expandir estados redundantes.

Exercício sugerido

Implemente o 8-puzzle utilizando o `generic_graph_search` da Seção 3.4:

- Defina `successors(state)` conforme o código acima.
- Use `is_goal(state)` para detectar o estado final.
- Compare o número de nós expandidos para **BFS** e **UCS**.

6 Exemplo: O Problema dos Missionários e Canibais

O **problema dos missionários e canibais** é um exemplo clássico de *busca em espaço de estados*, ilustrando como restrições e ações reversíveis afetam a estrutura do problema.

Descrição informal

Três missionários e três canibais precisam atravessar um rio usando um barco que comporta no máximo duas pessoas. A condição é que, em nenhum dos lados do rio, o número de canibais pode ser maior que o número de missionários (quando houver missionários presentes), pois, caso contrário, os canibais comeriam os missionários.

O objetivo é encontrar uma sequência de travessias que leve todos para a margem oposta de forma segura.

Representação de estados

Cada estado é representado como uma tupla:

$$(M_E, C_E, B)$$

onde:

- M_E é o número de missionários na margem esquerda;
- C_E é o número de canibais na margem esquerda;
- $B \in \{E, D\}$ indica a posição do barco (esquerda ou direita).

Como o total de pessoas é constante ($3M$ e $3C$), a situação da margem direita é deduzida automaticamente:

$$M_D = 3 - M_E, \quad C_D = 3 - C_E$$

Estado inicial e meta

$$s_0 = (3, 3, E), \qquad G = \{(0, 0, D)\}$$

Ações possíveis

O barco pode levar uma ou duas pessoas, em qualquer combinação válida de missionários e canibais:

$$A = \{(2, 0), (0, 2), (1, 1), (1, 0), (0, 1)\}$$

Cada ação (m, c) representa levar m missionários e c canibais para a outra margem.

Restrições de segurança

Um estado (M_E, C_E, B) é **válido** se:

$$\begin{cases} 0 \le M_E, C_E \le 3, \\ (M_E = 0 \text{ ou } M_E \ge C_E), \\ (M_D = 0 \text{ ou } M_D \ge C_D). \end{cases}$$

Função de transição

Dado o estado atual $s = (M_E, C_E, B)$ e a ação (m, c), o próximo estado s' é:

$$s' = \begin{cases} (M_E - m,\ C_E - c,\ D), & \text{se } B = E, \\ (M_E + m,\ C_E + c,\ E), & \text{se } B = D. \end{cases}$$

O novo estado é aceito apenas se for *válido* segundo as restrições acima.

Espaço de busca

O número máximo de estados possíveis é limitado:

$$|S| = 4 \times 4 \times 2 = 32,$$

mas vários estados são inválidos e são descartados pelas restrições de segurança.

Etapa	**Estado Atual** (M_E, C_E, B)	**Ação** (m, c)	**Novo Estado** (M'_E, C'_E, B')
1	(3,3,E)	(0,2)	(3,1,D)
2	(3,1,D)	(0,1)	(3,2,E)
3	(3,2,E)	(2,0)	(1,2,D)
4	(1,2,D)	(1,1)	(2,3,E)
5	(2,3,E)	(0,2)	(2,1,D)

Tabela 1: Sequência inicial de travessias válidas.

Exemplo de primeiras transições

Observações

- O problema ilustra a diferença entre **tree search** e **graph search**: muitas ações são reversíveis e podem gerar ciclos.
- A busca em largura (BFS) encontra a solução de menor número de travessias; a busca em profundidade (DFS) pode se prender em ciclos.
- O problema também pode ser visto como um **problema de satisfação de restrições** (CSP), onde cada margem deve obedecer às condições de segurança.

Exercício sugerido

Formalize o problema dos missionários e canibais em Python, implementando:

- Uma função `successors(state)` que gere todos os estados válidos.
- Uma função `is_goal(state)`.
- Aplique o algoritmo genérico de busca (Seção 3.4) com BFS para encontrar a sequência de ações mínima.

7 Graph Searching

Tree search vs Graph search

- **Tree search**: ignora estados repetidos; pode reexpandir o mesmo estado inúmeras vezes.
- **Graph search**: mantém *explored set* (fechados) e/ou *visited* para evitar repetições e ciclos.

Estados repetidos Sem controle de repetição, a complexidade explode. Em ambientes com *reversibility* (ações reversíveis), ciclos são comuns.

8 Uninformed Search Strategies (3.5)

Estratégias e Estruturas de Dados da Fronteira

- **BFS** (em largura): fila FIFO.
- **DFS** (em profundidade): pilha LIFO.
- **DLS** (depth-limited search): pilha com limite L.
- **IDS** (iterative deepening): repete DLS para $L = 0, 1, 2, \ldots$.
- **Uniform-Cost** (UCS): fila de prioridade por $g(n)$.

8.1 Implementações básicas

Busca em Largura (BFS) A **busca em largura** expande os nós nível a nível, utilizando uma fila (*FIFO*) para a fronteira. É completa e ótima se todos os passos têm custo igual.

```
from collections import deque

def add_fifo(frontier, node):
    """Adiciona o no ao final da fila (FIFO)."""
    if frontier is None:
        frontier = deque()
    frontier.append(node)
    return frontier

def pop_fifo(frontier):
    """Remove o no mais antigo da fila."""
    return frontier.popleft(), frontier
```

Busca em Profundidade (DFS) A **busca em profundidade** expande sempre o nó mais recentemente inserido na fronteira (pilha, ou *LIFO*). Usa pouca memória, mas pode não ser completa em espaços infinitos.

```python
def add_lifo(frontier, node):
    """Adiciona o no ao topo da pilha (LIFO)."""
    if frontier is None:
        frontier = []
    frontier.append(node)
    return frontier

def pop_lifo(frontier):
    """Remove o no mais recente da pilha."""
    return frontier.pop(), frontier
```

Busca de Custo Uniforme (UCS) A **busca de custo uniforme** (Uniform-Cost Search) expande sempre o nó com menor custo acumulado $g(n)$. A fronteira é implementada como uma fila de prioridade (`heapq`).

```python
import heapq

def add_priority_by_g(frontier, node):
    """Adiciona o no a fila de prioridade segundo o custo acumulado g(n)."""
    if frontier is None
        frontier = []
    heapq.heappush(frontier, (node.g, id(node), node))
    return frontier

def pop_priority(frontier):
    """Remove o no de menor custo acumulado g(n)."""
    _, _, node = heapq.heappop(frontier)
    return node, frontier
```

Resumo conceitual

- **BFS**: expande nós por profundidade — encontra o caminho mais curto em número de passos.
- **DFS**: explora ramos inteiros antes de retroceder — economiza memória, mas pode se perder em loops.
- **UCS**: expande por custo — ótima quando custos variam entre ações.

Tree vs Graph Search e Ótimalidade

- **BFS/IDS**: ótima em *tree* quando custo de passo é uniforme; em *graph*, manter visitados evita revisitar estados.
- **UCS**: ótima (com $c \geq \epsilon$) tanto em árvore quanto em grafo, desde que se gerencie duplicatas preservando o melhor g conhecido por estado.

Boas práticas

- **Fechados** (explored set) + **frontier** determinística.
- Tabela $best\_g[state]$ para descartar caminhos piores (UCS).
- Separar *estado* da *descrição do nó* (pai, ação, custos).

9 Propriedades comparativas

As estratégias de busca diferem quanto à **completude** (se garantem encontrar uma solução quando ela existe), **otimalidade** (se encontram a solução de menor custo), e **complexidade de tempo e espaço**.

Assuma:

- b — fator de ramificação (número médio de sucessores por nó);
- d — profundidade da solução mais rasa;
- m — profundidade máxima da árvore de busca (pode ser infinita);
- C — custo da solução ótima;
- ϵ — menor custo positivo de ação ($c(s, a, s') \geq \epsilon > 0$).

Observações complementares

- **Tree search** e **graph search** podem ter comportamento diferente: em *graph search*, é essencial registrar estados visitados para evitar reexploração e ciclos.
- **BFS** e **IDS** são ótimas apenas quando todos os custos de passo são iguais.
- **UCS** é ótima para custos não uniformes, desde que se mantenha o melhor custo $g(s)$ conhecido por estado.
- A escolha da estratégia depende de restrições práticas: memória disponível, profundidade esperada da solução e presença de ciclos.

Resumo didático

- BFS — *segura, mas cara* (ótima e completa, porém consome muita memória).
- DFS — *econômica, mas arriscada* (rápida e simples, mas pode falhar).
- IDS — *equilíbrio entre ambas.*
- UCS — *melhor escolha* quando há custos variados e é necessária otimalidade real.

Estratégia	Completa	Ótima	Tempo	Espaço	Observações
BFS	Sim (se b finito)	Sim (custos iguais)	$O(b^d)$	$O(b^d)$	Expande por profundidade; encontra solução mais rasa; alto custo de memória.
DFS	Não (em geral)	Não	$O(b^m)$	$O(b \cdot m)$	Pode entrar em loops; eficiente em memória; útil para espaços muito profundos.
DLS (limite L)	Não (se $L < d$)	Não	$O(b^L)$	$O(b \cdot L)$	Busca limitada em profundidade; útil quando há limite natural de profundidade.
IDS	Sim	Sim (custos iguais)	$O(b^d)$	$O(b \cdot d)$	Combina completude e ótimo da BFS com o uso de memória da DFS.
UCS	Sim ($c \geq \epsilon$)	Sim	$O\big(b^{1+\lfloor C/\epsilon \rfloor}\big)$	$\geq O(b^d)$	Expande por custo crescente; ótima mesmo com custos diferentes; requer controle de duplicatas.

Tabela 2: Propriedades comparativas das estratégias de busca não informadas (baseado em Poole & Mackworth, AIFCA 3e, Seção 3.5).

Mini-Exercícios (para fixação)

1. Modele o *Missionários e Canibais* como busca: defina S, A, γ, c, s_0, G.
2. Explique a diferença prática entre tree e graph search.
3. Implemente UCS em um grid $N \times N$ e compare com BFS.
4. Explique por que IDS repete trabalho mas ainda é eficiente.

Leitura Recomendada (AIFCA 3e)

- Cap. 2: Agentes e ambientes.
- Cap. 3.1–3.5: Problemas de busca, espaços de estados e estratégias não-informadas.