Roteiro de Aula 3 – Busca Informada (Cap. 3.5.4 e 3.6, Poole & Mackworth, AIFCA 3e)

Disciplina: BCC740 – Inteligência Artificial

20 de outubro de 2025

Sumário da Aula

1 Pré-requistos

1.1 Estruturas de Dados: Filas de Prioridade e Heaps

Os algoritmos **Uniform-Cost**, **AA***, e outras variantes de busca informada utilizam uma **fronteira** (ou *open list*) organizada como uma **fila de prioridade**. O objetivo é sempre expandir primeiro o nó com menor valor de custo (ou estimativa de custo).

Heaps: Estrutura de Dados para Fila de Prioridade

Uma implementação eficiente de fila de prioridade é o **heap binário**. Um heap é uma estrutura de árvore quase completa que mantém a seguinte propriedade:

> Para cada nó, o valor armazenado é menor ou igual (em um *min-heap*) ou maior ou igual (em um *max-heap*) que o valor dos filhos.

Isso garante que o elemento de menor valor esteja sempre na raiz, o que permite:

- Inserção de um novo elemento em tempo $O(\log n)$;
- Remoção (ou acesso) ao elemento mínimo em tempo $O(\log n)$;
- Verificação do mínimo em tempo $O(1)$.

Heap em Python

O módulo `heapq` da biblioteca padrão implementa um *min-heap* sobre listas. Os elementos inseridos devem ser *tuplas* ou objetos comparáveis, de modo que a comparação seja feita pelo primeiro item da tupla.

```python
import heapq

# Exemplo básico:
fronteira = []
heapq.heappush(fronteira, (3, "C"))
heapq.heappush(fronteira, (1, "A"))
heapq.heappush(fronteira, (2, "B"))

# Removendo o menor elemento
_, estado = heapq.heappop(fronteira)
print(estado)  # Saída: "A"
```

Na implementação de busca, as tuplas costumam conter:

$$(\text{prioridade}, \text{desempate}, \text{nó})$$

onde `prioridade` é o valor de custo $g(n)$ (UCS) ou $f(n) = g(n) + h(n)$ (AA*).

Resumo:

- `heapq.heappush(heap, elemento)` insere com custo $O(\log n)$;
- `heapq.heappop(heap)` remove o menor elemento com custo $O(\log n)$;
- Heaps são cruciais para manter a eficiência de UCS e AA*, evitando varreduras lineares.

2 Busca Informada: Motivação

As estratégias não informadas (BFS, DFS, UCS) não consideram diretamente quão perto um estado está da meta. A **busca informada** utiliza uma **função heurística** $h(n) \geq 0$ que estima o custo do melhor caminho de n até a meta.

Definições

h é **admissível** se $h(n) \leq h^*(n)$, onde $h^*(n)$ é o custo real mínimo de n à meta. Quanto melhor a qualidade de h, mais a busca é guiada para regiões promissoras do espaço.

Derivação de heurísticas (ideia): resolver um *problema simplificado* e usar seu custo como estimativa (linha reta em rotas; Manhattan no 8-puzzle; bancos de padrões etc.).

3 3.5.4 Lowest-Cost-First (Uniform-Cost) Search

Em muitos domínios, os arcos têm **custos não unitários** e o objetivo é **minimizar o custo total** do caminho até a meta. A **lowest-cost-first search** (ou **uniform-cost search**, UCS) expande sempre o caminho de **menor custo acumulado** $g(p)$ na fronteira.

Ideia e Implementação

- Trate a **fronteira** como uma **fila de prioridade** ordenada por $g(p)$.
- Ao expandir, gere sucessores com seus custos atualizados e reinsira na fronteira.
- Quebra de empates pode seguir ordem FIFO, menor profundidade, ou mesmo menor h (se disponível) apenas como *desempate* — não altera a otimalidade de UCS.

```python
import heapq

class Node:
    __slots__ = ("state","parent","action","g","depth")
    def __init__(self, state, parent=None, action=None, g=0.0):
        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):
    path = []
    while n and n.parent is not None:
        path.append(n.action)
        n = n.parent
    return list(reversed(path))

def uniform_cost_search(s0, is_goal, successors):
    """Retorna a sequência de ações de menor custo (se existir)."""
    start = Node(s0, None, None, g=0.0)
    frontier = [(0.0, id(start), start)]   # (g, tie, node)
    best_g = {}                            # melhor custo por estado

    while frontier:
        g, _, node = heapq.heappop(frontier)
        if is_goal(node.state):
            return solution(node)
        # poda por custo já conhecido
        if node.state in best_g and best_g[node.state] <= g:
            continue
        best_g[node.state] = g
        for (a, s_next, cost) in successors(node.state):
            g_next = g + float(cost)       # custos de arco > 0
            child = Node(s_next, node, a, g_next)
            heapq.heappush(frontier, (g_next, id(child), child))
    return None
```

Corretude e Condições Suficientes

Se (i) o fator de ramificação é finito e (ii) todos os custos de arco são **estritamente positivos** ($\geq \epsilon > 0$), então a **busca de custo uniforme (UCS)** é **completa** e **ótima**. Em particular, o *primeiro* caminho-meta removido da fronteira é aquele de menor custo total $g(p)$ — logo, a solução retornada é ótima.

Racional: a UCS expande caminhos a partir do nó inicial em *ordem crescente de custo acumulado*. Como os custos de arco são positivos, os valores de $g(p)$ nunca diminuem ao longo do processo. Assim, nenhum caminho mais barato pode ser descoberto após a expansão de um mais caro: se existisse uma solução de custo menor que a primeira encontrada, ela teria sido expandida antes da solução atual.

Importância do limite inferior ($\epsilon > 0$): A exigência de custos de arco *limitados inferiormente* — isto é, todos os arcos possuem custo maior que uma constante positiva — garante que o algoritmo não ficará preso em caminhos infinitos de custo finito. Com esse limite, a UCS sempre encontrará uma solução (se existir) em grafos com fator de ramificação finito.

Nota (paradoxo de Zeno): Sem o limite inferior ϵ, podem existir caminhos infinitos cujo custo total permanece finito. Por exemplo, considere uma sequência de nós $n_0, n_1, n_2, \ldots$ com arcos $\langle n_{i-1}, n_i \rangle$ de custo $1/2^i$. Cada caminho finito $\langle n_0, n_1, \ldots, n_k \rangle$ terá custo menor que 1, mas a soma total converge para 1. Se existir um arco direto de n_0 para o nó-meta com custo exatamente 1, a UCS nunca o expandirá, pois continuará gerando extensões infinitas do caminho parcial de custo ligeiramente menor que 1. Essa situação é uma analogia moderna ao *paradoxo de Zeno* descrito por Aristóteles há mais de dois milênios.

Complexidade (pior caso)

Se c é o custo da solução ótima e $k = c/\epsilon$, a busca pode gerar até $O(b^k)$ caminhos de comprimento $\leq k$, resultando em **tempo** e **espaço** $O(k\, b^k)$.

Ponte para A* UCS é um caso particular de A* quando $h(n) \equiv 0$, logo $f(n) = g(n)$. Essa visão unifica UCS e A* sob a mesma política de fila de prioridade.

4 Heuristic Depth-First e Greedy Best-First

Heuristic Depth-First Search

Ordena vizinhos por $h(n)$, expandindo o mais promissor primeiro. Não garante completude nem otimalidade.

Greedy Best-First Search

Seleciona sempre o nó com menor $h(n)$ na fronteira ($f(n) = h(n)$). Pode ser rápido, mas é míope e pode ciclar.

```
def greedy_best_first_search(s0, is_goal, successors, h):
    from heapq import heappush, heappop
    frontier = []
    heappush(frontier, (h(s0), Node(s0)))
    explored = set()
    while frontier:
        _, node = heappop(frontier)
        if is_goal(node.state):
            return solution(node)
        explored.add(node.state)
        for (a, s_next, cost) in successors(node.state):
            if s_next not in explored:
                heappush(frontier, (h(s_next), Node(s_next, node, a)))
    return None
```

5 Busca A*

A A* usa $f(n) = g(n) + h(n)$, combinando custo acumulado e heurística. Se h é admissível e os custos de arco são $\geq \epsilon > 0$, A* é **completa** e **ótima**.

```
def a_star_search(s0, is_goal, successors, h):
    import heapq
    start = Node(s0, None, None, 0.0)
    frontier = [(h(s0), id(start), start)]
    best_g = {}
    while frontier:
        f, _, node = heapq.heappop(frontier)
        if is_goal(node.state):
            return solution(node)
        if node.state in best_g and best_g[node.state] <= node.g:
            continue
        best_g[node.state] = node.g
        for (a, s_next, cost) in successors(node.state):
            g_next = node.g + float(cost)
            heapq.heappush(frontier, (g_next + h(s_next),
                                      id(node) ^ hash(s_next),
                                      Node(s_next, node, a, g_next)))
    return None
```

Intuição:

$$\text{start} \xrightarrow{g(n)} n \xrightarrow{h(n)} \text{goal} \quad \Rightarrow \quad f(n) = g(n) + h(n).$$

Admissibilidade (esboço): Com $b < \infty$, custos $\geq \epsilon > 0$ e h admissível, a primeira solução retornada por A* é ótima.

Consistência (Monotonicidade) da Heurística

Além de ser **admissível**, uma heurística pode também satisfazer uma condição mais forte, chamada de **consistência** (ou **monotonicidade**):

$$h(n) \leq c(n, n') + h(n')$$

para todo arco (n, n') com custo $c(n, n')$.

Intuição: a estimativa nunca "salta" mais do que o custo real entre dois estados adjacentes. Isso implica que a função $f(n) = g(n) + h(n)$ é *não decrescente* ao longo de qualquer caminho — o que permite reusar nós sem precisar reabrir a fronteira.

Relações Importantes:

- Toda heurística consistente é também admissível.
- O inverso nem sempre é verdadeiro.
- Se h é consistente, o A* não precisa reexpandir nós: o primeiro custo encontrado para um estado é o menor possível.

Prova (Esboço) de Otimalidade do A*

A seguir apresenta-se uma visão resumida da demonstração de que o A*, com h admissível e custos positivos, é **ótimo**.

Hipóteses:

1. Custos de arco $c(n, n') \geq \epsilon > 0$.
2. Fator de ramificação $b < \infty$.
3. Heurística h é admissível, ou seja, $h(n) \leq h^*(n)$.

Ideia da prova: Seja G_1 o primeiro nó-meta removido da fronteira pelo A*. Suponha, por contradição, que existe outro caminho P^* de custo menor ($C^* < g(G_1)$). Como h é admissível, cada nó n em P^* teria $f(n) = g(n) + h(n) \leq C^*$, logo todos esses nós teriam prioridade menor que G_1 na fila de prioridade. Mas então o A* teria expandido P^* antes de G_1, o que é uma contradição. Portanto, $g(G_1) = C^*$ e a solução é ótima.

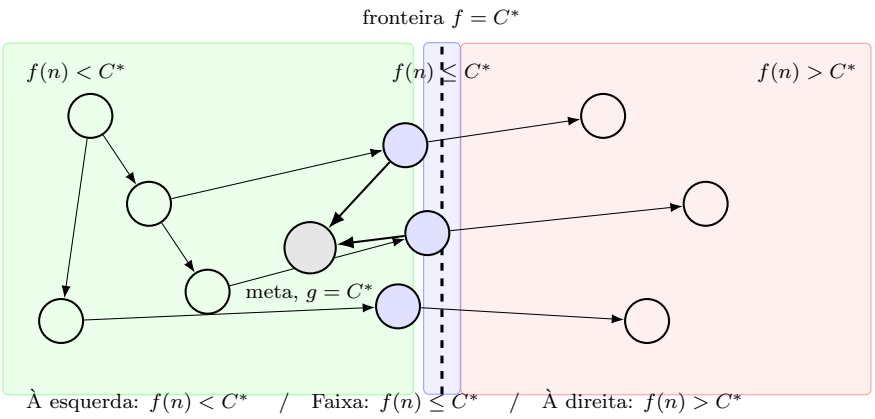

Figura 1: Esboço visual: os nós à esquerda da fronteira possuem $f(n) < C^*$, os da fronteira têm $f(n) \leq C^*$ e os à direita $f(n) > C^*$. O primeiro nó-meta removido terá custo ótimo C^*.

Visualização:

Consequência prática: A* é:

- **Completa**, se há limite inferior ϵ no custo dos arcos.
- **Ótima**, se h é admissível (ou consistente).

6 Branch and Bound

Combina **DFS** com um **limite (bound)**: poda caminhos com $g(n) + h(n)$ não menor que o melhor custo encontrado. Gera soluções cada vez melhores e retorna a ótima no final; economiza memória comparado a expandir em largura.

7 Construção de Heurísticas

Heurísticas admissíveis via *problema simplificado*:

- Rotas: distância em linha reta (Euclidiana).
- 8-puzzle: peças fora do lugar; soma de distâncias Manhattan.
- Entrega de pacotes: maior distância necessária entre coletas/destinos.

Pattern Databases: pré-compute soluções de subproblemas e armazene para consultas rápidas de h.

8 Resumo Comparativo

Método	**Usa $g(n)$**	**Usa $h(n)$**	**Completo**	**Ótimo**	**Observações**
UCS (Lowest-Cost-First)	Sim	Não	Sim ($\epsilon > 0$)	Sim	Caso especial de A* com $h \equiv 0$.
Greedy Best-First	Não	Sim	Não	Não	Rápido, mas míope; pode ciclar.
A*	Sim	Sim	Sim	Sim (se h admissível)	Equilíbrio entre custo real e estimado.
Branch and Bound	Sim	Opcional	Sim	Sim	DFS + poda por bound; economiza memória.

Tabela 1: Comparação entre métodos (assumindo custos de arco $\geq \epsilon > 0$ e ramificação finita).

Mini-Exercícios

1. Mostre por que UCS é um caso particular de A*.
2. Dê um exemplo em que BFS encontra o menor número de passos, mas UCS encontra um caminho de *menor custo*.
3. Modele um grafo pequeno e registre a fronteira a cada passo para UCS e A* com h admissível.
4. Projete uma heurística admissível para o 8-puzzle e compare A* com UCS.
5. Implemente branch and bound e compare com A* no mesmo problema.

Leitura Recomendada

- Poole & Mackworth, Seção 3.5.4 (UCS) e Seção 3.6 (Busca Informada).
- Seção 3.6.3: *Designing a Heuristic Function*.