Estratégias de Busca Heurística: A* e Gulosa

Empregando Algoritmos de Busca Informada para Otimizar a Resolução de Problemas

Márcio Nicolau

2025-10-09

Table of contents

List of Figures

Introdução

Nas aulas anteriores, exploramos as estratégias de busca cega (não-informada), como Busca em Largura (BFS) e Busca em Profundidade (DFS). Vimos que, embora sejam completas e, no caso da BFS, até ótimas sob certas condições, sua eficiência em espaços de estados grandes pode ser severamente limitada pela explosão combinatória. Elas não fazem uso de qualquer conhecimento sobre a "direção" do objetivo.

Nesta aula, introduziremos as **estratégias de busca heurística** (também conhecidas como busca informada). Esses algoritmos utilizam informações adicionais específicas do problema para guiar a busca de forma mais eficiente, otimizando o processo de encontrar uma solução. Focaremos em dois algoritmos proeminentes: a **Busca Gulosa (Greedy Best-First Search)** e a **Busca A* (A* Search)**.

Objetivo de Aprendizagem

Ao final desta aula, você será capaz de:

- Explicar as limitações dos algoritmos de busca cega e a motivação para a busca heurística.
- Definir o conceito de função heurística e suas propriedades.
- Compreender o funcionamento e aplicar o algoritmo de Busca Gulosa (Greedy Best-First Search).
- Compreender o funcionamento e aplicar o algoritmo de Busca A* (A* Search).
- Analisar as propriedades (completude, otimalidade, complexidade) de Busca Gulosa e A*.
- Distinguir entre os conceitos de heurísticas admissíveis e consistentes.
- Escolher a estratégia de busca informada mais apropriada para problemas específicos.

Problemas com Buscas Cegas (Não-Informadas): Uma Motivação

Algoritmos como BFS e DFS são métodos exaustivos que exploram o espaço de estados sem qualquer "inteligência" sobre onde o objetivo está.

- **BFS:** Garante o caminho mais curto em número de passos, mas expande todos os nós em cada nível. A complexidade de tempo e espaço é $O(b^d)$, o que se torna impraticável rapidamente para `d` (profundidade da solução) grande.
- **DFS:** Mais eficiente em memória ($O(bm)$), mas pode se perder em caminhos infinitos e não garante a otimalidade ou completude sem modificações. A complexidade de tempo também é $O(b^m)$ no pior caso.

Para muitos problemas do mundo real, o espaço de estados é vasto demais para ser explorado cegamente. Por exemplo, em um mapa de navegação com milhares de cidades ou em um jogo complexo, a busca cega seria ineficaz. Precisamos de uma maneira de direcionar a busca para as áreas mais promissoras do espaço de estados.

Heurísticas: Conhecimento para Guiar a Busca

A ideia central da busca informada é usar **conhecimento específico do problema** para guiar a exploração. Esse conhecimento é encapsulado em uma **função heurística**.

O que é uma Função Heurística ($h(n)$)?

Uma **função heurística**, denotada por $h(n)$, é uma estimativa do custo do caminho do nó `n` até o nó objetivo. Ela fornece uma "dica" de quão "próximo" um nó está do objetivo.

- $h(n) = 0$ se `n` é o nó objetivo.
- A heurística ideal seria a verdadeira distância do nó `n` ao objetivo, mas isso geralmente é impossível de saber sem resolver o problema.
- Uma boa heurística é aquela que fornece uma estimativa precisa sem ser muito custosa para calcular.
- **Exemplo:** No problema do mapa da Romênia (de Arad para Bucareste), uma heurística natural é a **distância em linha reta (straight-line distance)** de qualquer cidade até Bucareste. Esta é uma estimativa da distância real, mas é fácil de calcular.

Por que Heurísticas são Importantes?

Heurísticas permitem que os algoritmos de busca se concentrem em caminhos mais promissores, reduzindo drasticamente o número de nós que precisam ser expandidos e explorados. Isso transforma problemas intratáveis em problemas solucionáveis.

Busca Gulosa (Greedy Best-First Search - GBFS)

A Busca Gulosa é o mais simples dos algoritmos de busca informada. Ela expande o nó que está mais próximo do objetivo, de acordo com a função heurística $h(n)$.

Conceito e Funcionamento

A GBFS opera com a ideia de sempre seguir o caminho que *parece* ser o melhor no momento, sem considerar o custo já incorrido para chegar ao nó atual.

- **Função de Avaliação:** $f(n) = h(n)$
- **Estrutura de Dados:** Uma **fila de prioridade** (priority queue), que armazena os nós e os organiza com base no valor de $h(n)$, expandindo sempre o nó com o menor $h(n)$.

> **i** Algoritmo Básico da Busca Gulosa
>
> 1. Crie uma fila de prioridade e adicione o estado inicial a ela, com prioridade h(inicial).
> 2. Crie um conjunto para armazenar estados visitados.
> 3. Enquanto a fila de prioridade não estiver vazia:
>   a. Remova o nó com a menor prioridade ($h(n)$) da fila (o nó atual).

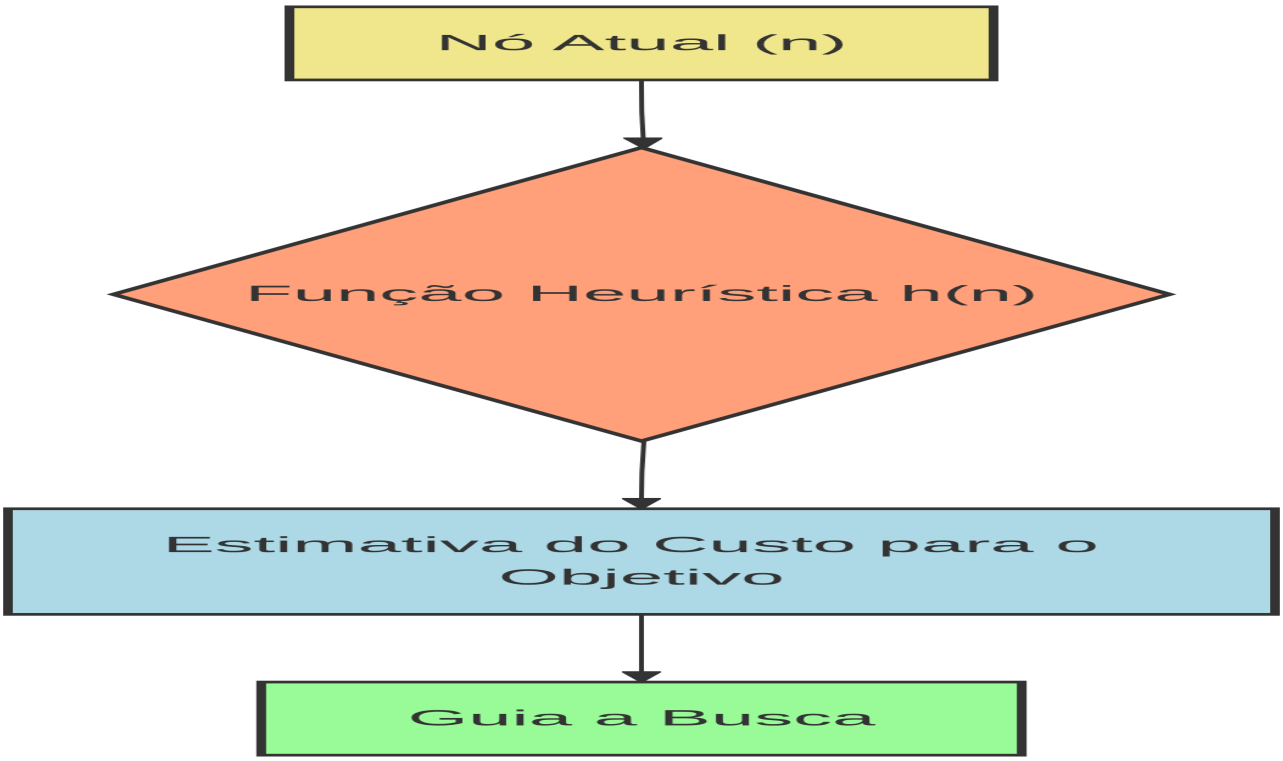

Figure 1: Conceito de Função Heurística

b. Se o nó atual for o estado objetivo, retorne o caminho para este nó.
c. Adicione o nó atual ao conjunto de visitados.
d. Para cada vizinho do nó atual:
    i. Se o vizinho não foi visitado e não está na fila:
        1. Adicione o vizinho à fila de prioridade, com prioridade h(vizinho).
        2. Marque o nó atual como pai do vizinho.

Exemplo Prático com Python (Mapa da Romênia)

Usaremos o mapa da Romênia e as distâncias em linha reta até Bucareste como heurística.

Cidade	Distância em linha reta até Bucareste
Arad	366
Bucareste	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
RimnicuVilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Implementação da Busca Gulosa em Python

```python
import heapq # Para fila de prioridade

# Mapa da Romênia (mesmo da aula de busca cega)
romania_map = {
    'Arad': {'Zerind': 75, 'Timisoara': 118, 'Sibiu': 140},
    'Zerind': {'Arad': 75, 'Oradea': 71},
    'Oradea': {'Zerind': 71, 'Sibiu': 151},
    'Timisoara': {'Arad': 118, 'Lugoj': 111},
    'Lugoj': {'Timisoara': 111, 'Mehadia': 70},
    'Mehadia': {'Lugoj': 70, 'Dobreta': 75},
    'Dobreta': {'Mehadia': 75, 'Craiova': 120},
    'Craiova': {'Dobreta': 120, 'RimnicuVilcea': 146, 'Pitesti': 138},
    'Sibiu': {'Arad': 140, 'Oradea': 151, 'Fagaras': 99, 'RimnicuVilcea': 80},
    'RimnicuVilcea': {'Sibiu': 80, 'Craiova': 146, 'Pitesti': 97},
    'Fagaras': {'Sibiu': 99, 'Bucuresti': 211},
    'Pitesti': {'RimnicuVilcea': 97, 'Craiova': 138, 'Bucuresti': 101},
    'Bucuresti': {'Fagaras': 211, 'Pitesti': 101, 'Giurgiu': 90, 'Urziceni': 85},
    'Giurgiu': {'Bucuresti': 90},
    'Urziceni': {'Bucuresti': 85, 'Hirsova': 98, 'Vaslui': 142},
    'Hirsova': {'Urziceni': 98, 'Eforie': 86},
    'Eforie': {'Hirsova': 86},
    'Vaslui': {'Urziceni': 142, 'Iasi': 92},
    'Iasi': {'Vaslui': 92, 'Neamt': 87},
    'Neamt': {'Iasi': 87}
}

# Heurística: Distância em linha reta até Bucareste
heuristic_straight_line_to_bucuresti = {
    'Arad': 366, 'Bucuresti': 0, 'Craiova': 160, 'Dobreta': 242, 'Eforie': 161,
    'Fagaras': 178, 'Giurgiu': 77, 'Hirsova': 151, 'Iasi': 226, 'Lugoj': 244,
    'Mehadia': 241, 'Neamt': 234, 'Oradea': 380, 'Pitesti': 98, 'RimnicuVilcea': 193,
    'Sibiu': 253, 'Timisoara': 329, 'Urziceni': 80, 'Vaslui': 199, 'Zerind': 374
}

def greedy_best_first_search(graph, start_node, goal_node, heuristic):
    # Fila de prioridade: (h(n), node, path)
    priority_queue = [(heuristic[start_node], start_node, [start_node])]
    visited = set()

    print(f"Iniciando Busca Gulosa de {start_node} para {goal_node}...")

    while priority_queue:
        current_h_cost, current_node, path = heapq.heappop(priority_queue)

        print(f"Expandindo nó: {current_node} (h={current_h_cost}). Caminho atual: {path}")

        if current_node == goal_node:
            print(f"Sucesso! Objetivo '{goal_node}' alcançado.")
            return path, current_h_cost # Retorna o caminho e o custo heurístico (não o custo real)

        if current_node not in visited:
            visited.add(current_node)

            for neighbor, cost in graph.get(current_node, {}).items():
                if neighbor not in visited:
```

Saída esperada da Busca Gulosa para cidades da Romênia

```
Iniciando Busca Gulosa de Arad para Bucuresti...
Expandindo nó: Arad (h=366). Caminho atual: ['Arad']
  Adicionando vizinho 'Zerind' (h=374) à fila de prioridade.
  Adicionando vizinho 'Timisoara' (h=329) à fila de prioridade.
  Adicionando vizinho 'Sibiu' (h=253) à fila de prioridade.
Expandindo nó: Sibiu (h=253). Caminho atual: ['Arad', 'Sibiu']
  Adicionando vizinho 'Oradea' (h=380) à fila de prioridade.
  Adicionando vizinho 'Fagaras' (h=178) à fila de prioridade.
  Adicionando vizinho 'RimnicuVilcea' (h=193) à fila de prioridade.
Expandindo nó: RimnicuVilcea (h=193). Caminho atual: ['Arad', 'Sibiu', 'RimnicuVilcea']
  Adicionando vizinho 'Craiova' (h=160) à fila de prioridade.
  Adicionando vizinho 'Pitesti' (h=98) à fila de prioridade.
Expandindo nó: Fagaras (h=178). Caminho atual: ['Arad', 'Sibiu', 'Fagaras']
  Adicionando vizinho 'Bucuresti' (h=0) à fila de prioridade.
Expandindo nó: Craiova (h=160). Caminho atual: ['Arad', 'Sibiu', 'RimnicuVilcea', 'Craiova']
  Adicionando vizinho 'Dobreta' (h=242) à fila de prioridade.
  Adicionando vizinho 'Pitesti' (h=98) à fila de prioridade.
Expandindo nó: Pitesti (h=98). Caminho atual: ['Arad', 'Sibiu', 'RimnicuVilcea', 'Pitesti']
  Adicionando vizinho 'Craiova' (h=160) à fila de prioridade.
  Adicionando vizinho 'Bucuresti' (h=0) à fila de prioridade.
Expandindo nó: Bucuresti (h=0). Caminho atual: ['Arad', 'Sibiu', 'Fagaras', 'Bucuresti']
Sucesso! Objetivo 'Bucuresti' alcançado.

Caminho encontrado de Arad para Bucuresti: Arad -> Sibiu -> Fagaras -> Bucuresti
Custo heurístico final (h(goal)): 0
```

Observe que a Busca Gulosa pode não encontrar o caminho mais curto em custo real. No exemplo, ele expandiu RimnicuVilcea e Craiova, seguindo a heurística, antes de encontrar Fagaras, que levou mais diretamente a Bucareste.

Propriedades da Busca Gulosa

- **Completude:** Não é completa. Em espaços de estados com caminhos infinitos ou ciclos, e sem um mecanismo para evitar repetição de nós (o conjunto `visited` ajuda aqui), pode se perder em um ramo sem fim, mesmo que o objetivo esteja em outro lugar. Se a heurística for ruim, pode ignorar o caminho correto.
- **Otimidade:** Não é ótima. Pode encontrar uma solução, mas não há garantia de que será o caminho de menor custo total. A "ganância" pode levar a atalhos que parecem bons heuristicamente, mas são caros na realidade. No exemplo acima, o caminho Arad -> Sibiu -> Fagaras -> Bucuresti (Custo total 140+99+211 = 450) foi encontrado, mas Arad -> Sibiu -> RimnicuVilcea -> Pitesti -> Bucuresti (Custo total 140+80+97+101 = 418) seria mais curto.
- **Complexidade de Tempo:** $O(b^m)$ no pior caso, onde `m` é a profundidade máxima. Pode ser significativamente melhor na prática se a heurística for boa.

- **Complexidade de Espaço:** $O(b^m)$ no pior caso, pois pode precisar armazenar todos os nós expandidos na fila de prioridade.

Busca A* (A* Search)

A Busca A* é o algoritmo de busca informada mais amplamente conhecido e utilizado. Ela combina o custo do caminho já percorrido até o nó atual ($g(n)$) com a estimativa do custo restante até o objetivo ($h(n)$).

Conceito e Funcionamento

A A* avalia cada nó n usando a função de avaliação $f(n) = g(n) + h(n)$:

- $g(n)$: Custo do caminho do nó inicial até o nó atual n. Este é o custo *real* e acumulado.
- $h(n)$: Custo estimado do caminho do nó atual n até o nó objetivo (a função heurística).
- $f(n)$: Custo total estimado do caminho mais barato do nó inicial, passando por n, até o nó objetivo.

A A* sempre expande o nó na fila de prioridade que possui o menor valor de $f(n)$.

Algoritmo Básico da Busca A*

1. Crie uma fila de prioridade e adicione o estado inicial a ela. A prioridade é $f(\text{inicial}) = g(\text{inicial}) + h(\text{inicial})$. Inicialmente, $g(\text{inicial}) = 0$.
2. Mantenha um mapa para armazenar o custo $g(n)$ para cada nó já descoberto (e o caminho para reconstruir).
3. Enquanto a fila de prioridade não estiver vazia:
    a. Remova o nó com a menor prioridade ($f(n)$) da fila (o nó atual).
    b. Se o nó atual for o estado objetivo, retorne o caminho reconstruído.
    c. Para cada vizinho do nó atual:
        i. Calcule $g(\text{vizinho}) = g(\text{atual}) + \text{custo da ação atual -> vizinho}$.
        ii. Se o vizinho ainda não foi visitado ou se o novo $g(\text{vizinho})$ é menor do que o anteriormente conhecido:
            1. Atualize $g(\text{vizinho})$.
            2. Calcule $f(\text{vizinho}) = g(\text{vizinho}) + h(\text{vizinho})$.
            3. Adicione (ou atualize) o vizinho na fila de prioridade com $f(\text{vizinho})$ como prioridade.
            4. Marque o nó atual como pai do vizinho.

Exemplo Prático com Python (Mapa da Romênia)

Usaremos o mesmo mapa e heurística (distância em linha reta) do exemplo da Busca Gulosa.

Implementação da Busca A* em Python

```python
import heapq

# Mapa da Romênia (com custos de aresta)
romania_map_weighted = {
    'Arad': {'Zerind': 75, 'Timisoara': 118, 'Sibiu': 140},
    'Zerind': {'Arad': 75, 'Oradea': 71},
    'Oradea': {'Zerind': 71, 'Sibiu': 151},
    'Timisoara': {'Arad': 118, 'Lugoj': 111},
    'Lugoj': {'Timisoara': 111, 'Mehadia': 70},
    'Mehadia': {'Lugoj': 70, 'Dobreta': 75},
    'Dobreta': {'Mehadia': 75, 'Craiova': 120},
    'Craiova': {'Dobreta': 120, 'RimnicuVilcea': 146, 'Pitesti': 138},
    'Sibiu': {'Arad': 140, 'Oradea': 151, 'Fagaras': 99, 'RimnicuVilcea': 80},
    'RimnicuVilcea': {'Sibiu': 80, 'Craiova': 146, 'Pitesti': 97},
    'Fagaras': {'Sibiu': 99, 'Bucuresti': 211},
    'Pitesti': {'RimnicuVilcea': 97, 'Craiova': 138, 'Bucuresti': 101},
    'Bucuresti': {'Fagaras': 211, 'Pitesti': 101, 'Giurgiu': 90, 'Urziceni': 85},
    'Giurgiu': {'Bucuresti': 90},
    'Urziceni': {'Bucuresti': 85, 'Hirsova': 98, 'Vaslui': 142},
    'Hirsova': {'Urziceni': 98, 'Eforie': 86},
    'Eforie': {'Hirsova': 86},
    'Vaslui': {'Urziceni': 142, 'Iasi': 92},
    'Iasi': {'Vaslui': 92, 'Neamt': 87},
    'Neamt': {'Iasi': 87}
}

# Heurística: Distância em linha reta até Bucareste
heuristic_straight_line_to_bucuresti = {
    'Arad': 366, 'Bucuresti': 0, 'Craiova': 160, 'Dobreta': 242, 'Eforie': 161,
    'Fagaras': 178, 'Giurgiu': 77, 'Hirsova': 151, 'Iasi': 226, 'Lugoj': 244,
    'Mehadia': 241, 'Neamt': 234, 'Oradea': 380, 'Pitesti': 98, 'RimnicuVilcea': 193,
    'Sibiu': 253, 'Timisoara': 329, 'Urziceni': 80, 'Vaslui': 199, 'Zerind': 374
}

def reconstruct_path(came_from, current_node):
    path = []
    while current_node is not None:
        path.append(current_node)
        current_node = came_from.get(current_node)
    return path[::-1] # Inverter o caminho para começar do início

def a_star_search(graph, start_node, goal_node, heuristic):
    # Fila de prioridade: (f(n), node, g(n))
    # f(n) é a prioridade, node é o nó, g(n) é o custo do início até o nó atual
    priority_queue = [(heuristic[start_node], start_node, 0)] # f_initial = g_initial + h_initial = 0

    # g_scores[n] é o custo do caminho mais barato conhecido do start_node para n
    g_scores = {start_node: 0}

    # came_from[n] é o nó precedente no caminho mais barato conhecido do start_node para n
    came_from = {start_node: None}
```

Saída esperada da Busca A* para cidades da Romênia

```
Iniciando Busca A* de Arad para Bucuresti...
Expandindo nó: Arad (g=0, h=366, f=366). Caminho atual: ['Arad']
  Adicionando/Atualizando vizinho 'Zerind' (g=75, h=374, f=449) na fila de prioridade.
  Adicionando/Atualizando vizinho 'Timisoara' (g=118, h=329, f=447) na fila de prioridade.
  Adicionando/Atualizando vizinho 'Sibiu' (g=140, h=253, f=393) na fila de prioridade.
Expandindo nó: Sibiu (g=140, h=253, f=393). Caminho atual: ['Arad', 'Sibiu']
  Adicionando/Atualizando vizinho 'Oradea' (g=291, h=380, f=671) na fila de prioridade.
  Adicionando/Atualizando vizinho 'Fagaras' (g=239, h=178, f=417) na fila de prioridade.
  Adicionando/Atualizando vizinho 'RimnicuVilcea' (g=220, h=193, f=413) na fila de prioridade.
Expandindo nó: RimnicuVilcea (g=220, h=193, f=413). Caminho atual: ['Arad', 'Sibiu', 'RimnicuVilcea']
  Adicionando/Atualizando vizinho 'Craiova' (g=366, h=160, f=526) na fila de prioridade.
  Adicionando/Atualizando vizinho 'Pitesti' (g=317, h=98, f=415) na fila de prioridade.
Expandindo nó: Fagaras (g=239, h=178, f=417). Caminho atual: ['Arad', 'Sibiu', 'Fagaras']
  Adicionando/Atualizando vizinho 'Bucuresti' (g=450, h=0, f=450) na fila de prioridade.
Expandindo nó: Pitesti (g=317, h=98, f=415). Caminho atual: ['Arad', 'Sibiu', 'RimnicuVilcea', 'Pites
  Vizinho 'RimnicuVilcea' já na fila ou visitado com custo menor/igual. Ignorando.
  Vizinho 'Craiova' já na fila ou visitado com custo menor/igual. Ignorando.
  Adicionando/Atualizando vizinho 'Bucuresti' (g=418, h=0, f=418) na fila de prioridade.
Expandindo nó: Bucuresti (g=418, h=0, f=418). Caminho atual: ['Arad', 'Sibiu', 'RimnicuVilcea', 'Pites
Sucesso! Objetivo 'Bucuresti' alcançado.

Caminho encontrado de Arad para Bucuresti: Arad -> Sibiu -> RimnicuVilcea -> Pitesti -> Bucuresti
Custo total do caminho: 418
```

Observe que a A encontrou o caminho mais curto (418), diferentemente da Busca Gulosa que encontrou um caminho de custo 450. Isso demonstra a otimalidade da A* com uma heurística admissível.*

Propriedades da Busca A* (Russell; Norvig, 2004)

- **Completude:** Sim, se o fator de ramificação `b` for finito e os custos das arestas forem maiores que zero.
- **Otimidade:** Sim, se a heurística $h(n)$ for **admissível** (nunca superestima o custo real até o objetivo) e os custos das arestas forem não-negativos. Se a heurística for também **consistente** (condição mais forte), a A* não precisa re-explorar nós.
- **Complexidade de Tempo:** Pode ser $O(b^d)$ no pior caso, mas na prática é significativamente mais eficiente do que as buscas cegas. O desempenho real depende da qualidade da heurística.
- **Complexidade de Espaço:** $O(b^d)$ no pior caso, pois precisa armazenar todos os nós gerados na fila de prioridade (e os `g_scores` e `came_from`). Esta é sua principal desvantagem em problemas com espaços de estados muito grandes.

Admissibilidade e Consistência de Heurísticas

Para garantir a otimalidade da Busca A*, a heurística utilizada deve satisfazer certas propriedades.

Admissibilidade

Uma função heurística $h(n)$ é **admissível** se ela nunca superestima o custo real de chegar ao objetivo a partir do nó `n`. Formalmente:

- $h(n) \leq h^*(n)$ para todo nó `n`, onde $h^*(n)$ é o custo real do caminho mais barato de `n` ao objetivo.
- **Importância:** A admissibilidade é crucial para garantir a **otimalidade** da Busca A*. Se A* usa uma heurística admissível, ela sempre encontrará o caminho de menor custo, assumindo custos de aresta não-negativos.
- **Exemplo:** A distância em linha reta é uma heurística admissível para problemas de navegação em mapas (como o da Romênia), pois a menor distância entre dois pontos é sempre uma linha reta, e não podemos viajar mais rápido do que isso.

Consistência (ou Monotonicidade)

Uma função heurística $h(n)$ é **consistente** se, para todo nó `n` e qualquer sucessor `n'` de `n` gerado por uma ação com custo $c(n, n')$:

- $h(n) \leq c(n, n') + h(n')$
- **Importância:** A consistência é uma condição mais forte que a admissibilidade. Se uma heurística é consistente, ela também é admissível. Em algoritmos como A*, uma heurística consistente garante que o valor de $f(n)$ nunca diminui ao longo de um caminho. Isso evita a necessidade de reabrir e re-expandir nós já expandidos, simplificando a implementação e garantindo que o primeiro caminho encontrado para qualquer nó é o caminho mais curto para aquele nó.

Considerações Práticas para Busca Heurística

- **Design de Heurísticas:** A qualidade da heurística é crucial. Uma heurística mal projetada pode fazer com que a busca informada se comporte como uma busca cega ou, pior, leve a soluções não-ótimas (se não for admissível) ou a um desempenho muito ruim.
  - Muitas vezes, heurísticas são derivadas de versões relaxadas do problema, onde algumas restrições são removidas para tornar o problema mais fácil de estimar (e o custo real nunca é menor que o custo relaxado).
- **Trade-offs:**
  - **Busca Gulosa:** Rápida para encontrar *alguma* solução, mas não garante a melhor e pode se perder. Boa quando a velocidade é mais importante que a otimalidade e a heurística é confiável.
  - **Busca A***: Garante a solução ótima (com heurística admissível), mas pode ser mais lenta e consumir mais memória que a Busca Gulosa se a heurística for muito conservadora (baixa estimativa). É o algoritmo preferencial quando otimalidade é um requisito.
- **Memória:** A principal limitação da A* é o consumo de memória ($O(b^d)$). Para problemas com espaços de estados enormes, variações como a A* Iterativa (IDA*) ou a A* de Memória Limitada (SMA*) são usadas para gerenciar o uso de memória.

Escolhendo a Melhor Estratégia

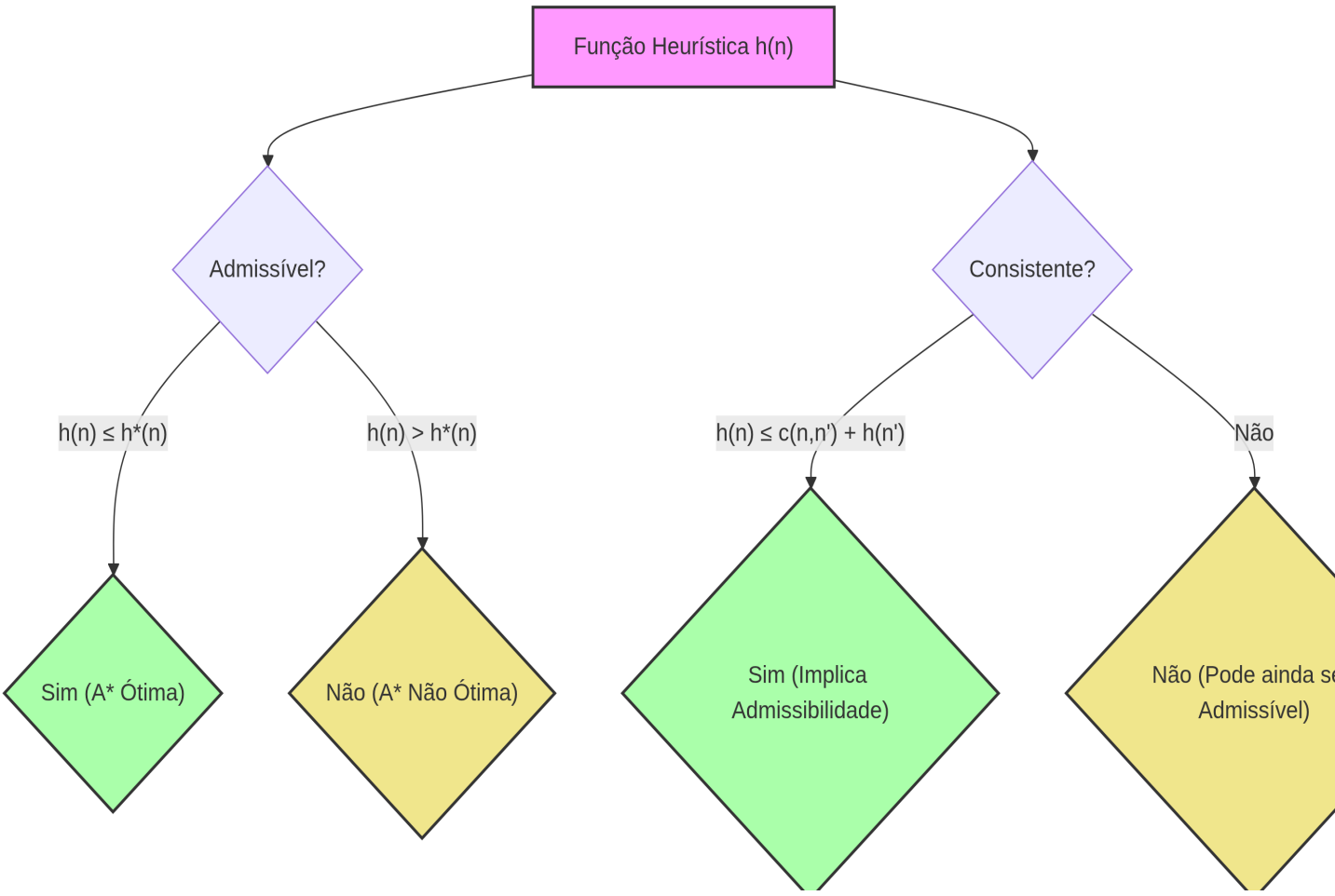

Figure 2: Propriedades de Funções Heurísticas

Problema / Cenário	Estratégia Recomendada
Encontrar *qualquer* caminho rápido.	DFS ou Busca Gulosa
Encontrar o *caminho mais curto* em termos de passos (custos uniformes).	BFS
Encontrar o *caminho de menor custo* total (custos não uniformes), com memória ilimitada.	Busca A*
Encontrar o *caminho de menor custo* total, com memória limitada.	A* Iterativa (IDA*) ou SMA*
Espaço de estados muito grande, heurística forte.	Busca Gulosa ou A*

Verificação de Aprendizagem

Responda às seguintes questões e implemente as tarefas propostas para solidificar seu entendimento.

1. **Motivação para Busca Informada:**
   a) Quais são as principais desvantagens da Busca em Largura (BFS) e Busca em Profundidade (DFS) em problemas com grandes espaços de estados?
   b) Como uma função heurística ($h(n)$) ajuda a superar essas desvantagens?
2. **Busca Gulosa:**
   a) Explique o princípio de funcionamento da Busca Gulosa. Qual função de avaliação ($f(n)$) ela utiliza?
   b) Por que a Busca Gulosa não é garantidamente ótima? Dê um exemplo (pode ser conceitual ou desenhado) onde ela escolheria um caminho subótimo.
3. **Busca A*:**
   a) Qual a função de avaliação ($f(n)$) utilizada pela Busca A*? Explique o significado de cada componente.
   b) Em qual aspecto principal a Busca A* se difere da Busca Gulosa, levando-a a ser ótima sob certas condições?
4. **Propriedades de Heurísticas:**
   a) Defina uma heurística **admissível**. Por que a admissibilidade é crucial para a otimalidade da Busca A*?
   b) Defina uma heurística **consistente**. Qual a relação entre consistência e admissibilidade?
5. **Implementação e Análise Comparativa:** Considere um problema de labirinto simples onde o agente precisa ir de um ponto inicial 'S' a um ponto final 'G'. O labirinto é representado por uma grade 2D, e o agente pode se mover para cima, baixo, esquerda ou direita (custo 1 por movimento).

```
[['S', ' ', ' ', ' '],
 [' ', 'X', ' ', ' '],
 [' ', ' ', ' ', ' '],
 [' ', ' ', 'X', 'G']]
```

   onde 'X' são obstáculos.
   a) Represente este labirinto como um grafo (nós são coordenadas `(linha, coluna)`) em Python.

b) Defina uma função heurística admissível e consistente para este labirinto (dica: distância de Manhattan).
c) Adapte a função `greedy_best_first_search` e `a_star_search` para resolver este labirinto.
d) Execute ambas as buscas de 'S' para 'G'. Compare:
   i. O caminho encontrado por cada algoritmo.
   ii. O custo total do caminho de cada algoritmo.
   iii. A ordem em que os nós foram expandidos (você pode inferir isso da saída de print).
   iv. Comente se os resultados confirmam as propriedades de otimalidade e eficiência esperadas para cada algoritmo neste cenário.

Referências Bibliográficas

RUSSELL, Stuart J.; NORVIG, Peter. **Inteligência Artificial: Um Enfoque Moderno**. 2. ed. Rio de Janeiro: Prentice Hall, 2004.