Estratégias de Busca: Busca em Largura e Profundidade

Aplicando Algoritmos de Busca Cega em Grafos

Márcio Nicolau

Table of contents

List of Figures

Introdução

No campo da Inteligência Artificial, muitos problemas podem ser formulados como a busca por uma sequência de ações que transformam um estado inicial em um estado objetivo. Imagine encontrar o caminho mais curto para um destino, resolver um quebra-cabeça, ou planejar os movimentos de um robô. Todos esses cenários podem ser modelados como problemas de busca. Nesta aula, exploraremos as estratégias de **busca cega** (também conhecidas como busca não-informada), focando em dois algoritmos fundamentais: a Busca em Largura (Breadth-First Search - BFS) e a Busca em Profundidade (Depth-First Search - DFS).

Objetivo de Aprendizagem

Ao final desta aula, você será capaz de:

- Formular problemas de busca em termos de estados, ações e metas.
- Compreender o funcionamento e aplicar os algoritmos de Busca em Largura (BFS) e Busca em Profundidade (DFS).
- Analisar as propriedades (completude, otimalidade, complexidade de tempo e espaço) de BFS e DFS.
- Identificar quando cada algoritmo é mais apropriado para a resolução de um problema.

Representação de Problemas para Busca

Antes de mergulharmos nos algoritmos, é crucial entender como um problema real é transformado em um problema de busca. A maioria dos problemas de busca em IA é modelada como a exploração de um **espaço de estados**, que pode ser visualizado como um grafo.

Elementos de um Problema de Busca

De acordo com Russell; Norvig (2004), um problema de busca é formalmente definido pelos seguintes componentes:

1. **Estado Inicial (Initial State):** O ponto de partida. Por exemplo, a localização atual de um carro em um mapa.
2. **Ações (Actions) ou Operadores:** Uma função que, dado um estado, retorna um conjunto de ações possíveis nesse estado. Para cada ação, há uma descrição de qual estado ela levará. Por exemplo, em um mapa, as ações seriam “mover para o norte”, “mover para o sul”, etc.
3. **Função de Transição de Estado (Transition Model):** Descreve o que cada ação faz. Formalmente, `RESULTADO(s, a)` retorna o estado resultante da execução da ação `a` no estado `s`.
4. **Teste de Meta (Goal Test):** Uma função que determina se um dado estado é um estado objetivo. Por exemplo, `É_OBJETIVO(cidade)` retorna `True` se a `cidade` for o destino desejado.
5. **Custo do Caminho (Path Cost):** Uma função que atribui um custo numérico a cada caminho. Normalmente, é a soma dos custos individuais de cada ação no caminho. O custo de uma ação pode ser a distância percorrida, tempo, combustível gasto, etc.

O conjunto de todos os estados alcançáveis a partir do estado inicial, juntamente com as ações que os conectam, forma o **espaço de estados** do problema. Um **caminho** é uma sequência de estados conectados por ações, e a solução para um problema de busca é um caminho do estado inicial a um estado objetivo.

Grafos e Árvores de Busca

Podemos representar o espaço de estados como um **grafo**, onde os nós são os estados e as arestas são as ações que levam de um estado a outro.

Para um algoritmo de busca, frequentemente construímos uma **árvore de busca**. Cada nó na árvore de busca corresponde a um estado no espaço de estados e também contém informações sobre o caminho percorrido para alcançá-lo (estado pai, ação que levou a ele, custo do caminho).

Algoritmos de Busca Cega (Uninformed Search)

Os algoritmos de busca cega, ou não-informada, são assim chamados porque não possuem nenhuma informação adicional sobre o custo ou a distância até o objetivo além da definição do problema em si. Eles operam de forma sistemática para explorar o espaço de estados até encontrar uma solução.

Propriedades de Algoritmos de Busca

Ao avaliar um algoritmo de busca, consideramos quatro propriedades principais (Russell; Norvig, 2004, p. 71):

1. **Completude (Completeness):** O algoritmo garante que encontrará uma solução se uma existir?
2. **Otimidade (Optimality):** O algoritmo garante que encontrará a *melhor* solução (aquela com o menor custo de caminho) se várias soluções existirem?
3. **Complexidade de Tempo (Time Complexity):** Quanto tempo o algoritmo leva para encontrar uma solução? Medido pelo número de nós gerados/expandidos.
4. **Complexidade de Espaço (Space Complexity):** Quanta memória o algoritmo precisa? Medido pelo número máximo de nós armazenados na memória.

As complexidades de tempo e espaço são geralmente expressas em termos de:

- `b`: fator de ramificação (branching factor) máximo (número máximo de ações possíveis a partir de qualquer estado).
- `d`: profundidade da solução mais rasa (menor número de ações para chegar a uma solução).
- `m`: profundidade máxima do espaço de estados.

Busca em Largura (Breadth-First Search - BFS)

A Busca em Largura é uma estratégia de busca que explora o espaço de estados camada por camada, visitando todos os nós em um determinado nível de profundidade antes de passar para o próximo nível. É como explorar um labirinto expandindo todas as opções que estão a uma passo de distância, depois todas as opções a dois passos, e assim por diante.

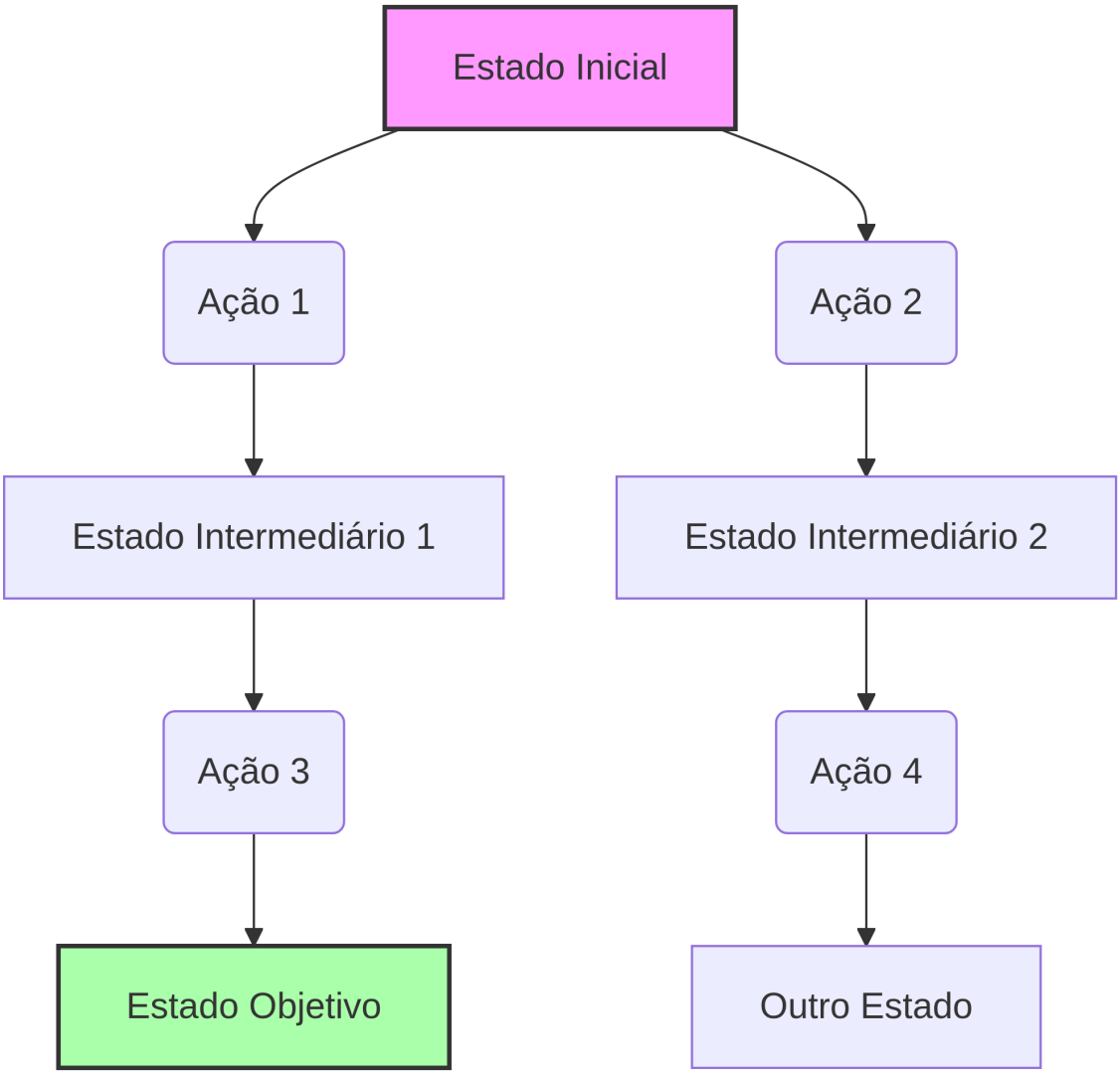

Figure 1: Representação de Problemas para Busca

Conceito e Funcionamento

A BFS garante que o nó objetivo mais raso (mais próximo do nó inicial em termos de número de passos) seja encontrado primeiro. Isso significa que, se as ações tiverem custo uniforme (ou seja, cada passo custa o mesmo), a BFS encontrará o caminho mais curto em número de passos.

Para implementar a BFS, utiliza-se uma **fila (queue)** (FIFO - First-In, First-Out) para armazenar os nós a serem visitados.

> **Algoritmo Básico**
>
> 1. Crie uma fila e adicione o estado inicial a ela.
> 2. Crie um conjunto para armazenar estados visitados (para evitar ciclos e reprocessamento).
> 3. Enquanto a fila não estiver vazia:
>   a. Remova o primeiro nó da fila (o nó atual).
>   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 (estado alcançável) do nó atual:
>   i. Se o vizinho não foi visitado e não está na fila:
>   1. Adicione o vizinho à fila.
>   2. Marque o nó atual como pai do vizinho (para reconstruir o caminho).

Exemplo Prático com Python

Vamos considerar um problema de busca em um mapa simplificado de cidades. Queremos encontrar um caminho de “Arad” para “Bucareste”.

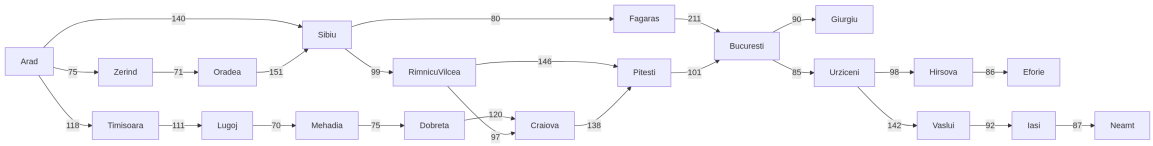

Figure 2: Representação de Problemas para Busca

Representação do Grafo em Python

```python
# graph_map.py
romania_map = {
    'Arad': ['Zerind', 'Timisoara', 'Sibiu'],
    'Zerind': ['Arad', 'Oradea'],
    'Oradea': ['Zerind', 'Sibiu'],
    'Timisoara': ['Arad', 'Lugoj'],
    'Lugoj': ['Timisoara', 'Mehadia'],
    'Mehadia': ['Lugoj', 'Dobreta'],
    'Dobreta': ['Mehadia', 'Craiova'],
    'Craiova': ['Dobreta', 'RimnicuVilcea', 'Pitesti'],
    'Sibiu': ['Arad', 'Oradea', 'Fagaras', 'RimnicuVilcea'],
    'RimnicuVilcea': ['Sibiu', 'Craiova', 'Pitesti'],
    'Fagaras': ['Sibiu', 'Bucuresti'],
    'Pitesti': ['RimnicuVilcea', 'Craiova', 'Bucuresti'],
    'Bucuresti': ['Fagaras', 'Pitesti', 'Giurgiu', 'Urziceni'],
    'Giurgiu': ['Bucuresti'],
    'Urziceni': ['Bucuresti', 'Hirsova', 'Vaslui'],
    'Hirsova': ['Urziceni', 'Eforie'],
    'Eforie': ['Hirsova'],
    'Vaslui': ['Urziceni', 'Iasi'],
    'Iasi': ['Vaslui', 'Neamt'],
    'Neamt': ['Iasi']
}
```

Saída esperada da BFS para cidades da Romênia

```
Iniciando BFS de Arad para Bucuresti...
Expandindo nó: Arad. Caminho atual: ['Arad']
  Adicionando vizinho 'Zerind' à fila. Fila atual: ['Zerind']
  Adicionando vizinho 'Timisoara' à fila. Fila atual: ['Zerind', 'Timisoara']
  Adicionando vizinho 'Sibiu' à fila. Fila atual: ['Zerind', 'Timisoara', 'Sibiu']
Expandindo nó: Zerind. Caminho atual: ['Arad', 'Zerind']
  Adicionando vizinho 'Oradea' à fila. Fila atual: ['Timisoara', 'Sibiu', 'Oradea']
Expandindo nó: Timisoara. Caminho atual: ['Arad', 'Timisoara']
  Adicionando vizinho 'Lugoj' à fila. Fila atual: ['Sibiu', 'Oradea', 'Lugoj']
Expandindo nó: Sibiu. Caminho atual: ['Arad', 'Sibiu']
  Adicionando vizinho 'Oradea' à fila. Fila atual: ['Oradea', 'Lugoj', 'Oradea']
  Adicionando vizinho 'Fagaras' à fila. Fila atual: ['Oradea', 'Lugoj', 'Oradea', 'Fagaras']
  Adicionando vizinho 'RimnicuVilcea' à fila. Fila atual: ['Oradea', 'Lugoj', 'Oradea', 'Fagaras', 'R
Expandindo nó: Oradea. Caminho atual: ['Arad', 'Zerind', 'Oradea']
Expandindo nó: Lugoj. Caminho atual: ['Arad', 'Timisoara', 'Lugoj']
```

```
  Adicionando vizinho 'Mehadia' à fila. Fila atual: ['Oradea', 'Fagaras', 'RimnicuVilcea', 'Mehadia']
Expandindo nó: Oradea. Caminho atual: ['Arad', 'Sibiu', 'Oradea']
  Nó 'Oradea' já visitado, ignorando.
Expandindo nó: Fagaras. Caminho atual: ['Arad', 'Sibiu', 'Fagaras']
  Adicionando vizinho 'Bucuresti' à fila. Fila atual: ['RimnicuVilcea', 'Mehadia', 'Bucuresti']
Expandindo nó: RimnicuVilcea. Caminho atual: ['Arad', 'Sibiu', 'RimnicuVilcea']
  Adicionando vizinho 'Craiova' à fila. Fila atual: ['Mehadia', 'Bucuresti', 'Craiova']
  Adicionando vizinho 'Pitesti' à fila. Fila atual: ['Mehadia', 'Bucuresti', 'Craiova', 'Pitesti']
Expandindo nó: Mehadia. Caminho atual: ['Arad', 'Timisoara', 'Lugoj', 'Mehadia']
  Adicionando vizinho 'Dobreta' à fila. Fila atual: ['Bucuresti', 'Craiova', 'Pitesti', 'Dobreta']
Expandindo nó: Bucuresti. Caminho atual: ['Arad', 'Sibiu', 'Fagaras', 'Bucuresti']
Sucesso! Objetivo 'Bucuresti' alcançado.

Caminho encontrado de Arad para Bucuresti: Arad -> Sibiu -> Fagaras -> Bucuresti
```

Observe que a saída real da execução do código ***`bfs_search.py`*** *mostrará o processo de expansão de nós e adicionará vizinhos à fila de forma mais detalhada, demonstrando a natureza "nível a nível" da busca.*

Propriedades da BFS

- **Completude:** Sim, se o fator de ramificação `b` for finito, a BFS é completa. Ela eventualmente encontrará o objetivo se ele existir, pois explora todos os nós a cada nível antes de ir mais fundo.
- **Otimidade:** Sim, se os custos das ações são uniformes (todos os custos de aresta são iguais), a BFS é ótima. Ela sempre encontra o caminho com o menor número de passos. Se os custos das arestas forem diferentes, a BFS não garante otimalidade; nesse caso, um algoritmo como a Busca de Custo Uniforme (Uniform-Cost Search) seria mais adequado.
- **Complexidade de Tempo:** $O(b^d)$, onde `b` é o fator de ramificação e `d` é a profundidade da solução mais rasa. Isso ocorre porque, no pior caso, a BFS pode ter que expandir todos os nós até a profundidade `d`.
- **Complexidade de Espaço:** $O(b^d)$, pois a fila pode ter que armazenar quase todos os nós no nível `d` da árvore de busca. Esta é a maior desvantagem da BFS em problemas com espaços de estados grandes.

Busca em Profundidade (Depth-First Search - DFS)

A Busca em Profundidade explora o espaço de estados indo o mais fundo possível ao longo de cada ramo antes de retroceder e tentar outro ramo. É como entrar em um labirinto, seguir um caminho até o fim (um beco sem saída ou o objetivo), e se não for o objetivo, voltar um passo e tentar outro caminho.

Conceito e Funcionamento

A DFS segue um caminho até seu final lógico (ou uma profundidade máxima definida) e só então retrocede (backtrack) para explorar outros caminhos.

Para implementar a DFS, utiliza-se uma **pilha (stack)** (LIFO - Last-In, First-Out) para armazenar os nós a serem visitados. Alternativamente, pode ser implementada recursivamente, onde a pilha é implicitamente gerenciada pelas chamadas de função do sistema.

Algoritmo Básico

1. Crie uma pilha e adicione o estado inicial a ela.
2. Crie um conjunto para armazenar estados visitados.
3. Enquanto a pilha não estiver vazia:
    a. Remova o nó do topo da pilha (o nó atual).
    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 (estado alcançável) do nó atual (em ordem reversa, se a ordem de expansão for importante para o problema, ou a ordem padrão do grafo):
        i. Se o vizinho não foi visitado e não está na pilha:
            1. Adicione o vizinho à pilha.
            2. Marque o nó atual como pai do vizinho (para reconstruir o caminho).

Exemplo Prático com Python

Usaremos o mesmo mapa da Romênia para demonstrar a DFS.

Saída esperada da DFS para cidades da Romênia

```
Iniciando DFS de Arad para Bucuresti com limite de profundidade infinito...
Expandindo nó: Arad (Profundidade: 0). Caminho atual: ['Arad']
  Adicionando vizinho 'Sibiu' à pilha. Pilha atual: ['Sibiu']
  Adicionando vizinho 'Timisoara' à pilha. Pilha atual: ['Sibiu', 'Timisoara']
  Adicionando vizinho 'Zerind' à pilha. Pilha atual: ['Sibiu', 'Timisoara', 'Zerind']
Expandindo nó: Zerind (Profundidade: 1). Caminho atual: ['Arad', 'Zerind']
  Adicionando vizinho 'Oradea' à pilha. Pilha atual: ['Sibiu', 'Timisoara', 'Oradea']
Expandindo nó: Oradea (Profundidade: 2). Caminho atual: ['Arad', 'Zerind', 'Oradea']
  Adicionando vizinho 'Sibiu' à pilha. Pilha atual: ['Sibiu', 'Timisoara', 'Sibiu']
Expandindo nó: Sibiu (Profundidade: 3). Caminho atual: ['Arad', 'Zerind', 'Oradea', 'Sibiu']
  Adicionando vizinho 'RimnicuVilcea' à pilha. Pilha atual: ['Sibiu', 'Timisoara', 'RimnicuVilcea']
  Adicionando vizinho 'Fagaras' à pilha. Pilha atual: ['Sibiu', 'Timisoara', 'RimnicuVilcea', 'Fagaras
Expandindo nó: Fagaras (Profundidade: 4). Caminho atual: ['Arad', 'Zerind', 'Oradea', 'Sibiu', 'Fagara
  Adicionando vizinho 'Bucuresti' à pilha. Pilha atual: ['Sibiu', 'Timisoara', 'RimnicuVilcea', 'Bucu
Expandindo nó: Bucuresti (Profundidade: 5). Caminho atual: ['Arad', 'Zerind', 'Oradea', 'Sibiu', 'Fag
Sucesso! Objetivo 'Bucuresti' alcançado.

Caminho encontrado de Arad para Bucuresti: Arad -> Zerind -> Oradea -> Sibiu -> Fagaras -> Bucuresti

--- Exemplo DFS com Limite de Profundidade ---
```

```
Iniciando DFS de Arad para Bucuresti com limite de profundidade 2...
Expandindo nó: Arad (Profundidade: 0). Caminho atual: ['Arad']
  Adicionando vizinho 'Sibiu' à pilha. Pilha atual: ['Sibiu']
  Adicionando vizinho 'Timisoara' à pilha. Pilha atual: ['Sibiu', 'Timisoara']
  Adicionando vizinho 'Zerind' à pilha. Pilha atual: ['Sibiu', 'Timisoara', 'Zerind']
Expandindo nó: Zerind (Profundidade: 1). Caminho atual: ['Arad', 'Zerind']
  Adicionando vizinho 'Oradea' à pilha. Pilha atual: ['Sibiu', 'Timisoara', 'Oradea']
Expandindo nó: Oradea (Profundidade: 2). Caminho atual: ['Arad', 'Zerind', 'Oradea']
  Limite de profundidade 2 alcançado em 'Oradea', retrocedendo.
Expandindo nó: Timisoara (Profundidade: 1). Caminho atual: ['Arad', 'Timisoara']
  Adicionando vizinho 'Lugoj' à pilha. Pilha atual: ['Sibiu', 'Lugoj']
Expandindo nó: Lugoj (Profundidade: 2). Caminho atual: ['Arad', 'Timisoara', 'Lugoj']
  Limite de profundidade 2 alcançado em 'Lugoj', retrocedendo.
Expandindo nó: Sibiu (Profundidade: 1). Caminho atual: ['Arad', 'Sibiu']
  Adicionando vizinho 'RimnicuVilcea' à pilha. Pilha atual: ['RimnicuVilcea']
  Adicionando vizinho 'Fagaras' à pilha. Pilha atual: ['RimnicuVilcea', 'Fagaras']
  Adicionando vizinho 'Oradea' à pilha. Pilha atual: ['RimnicuVilcea', 'Fagaras', 'Oradea']
Expandindo nó: Oradea (Profundidade: 2). Caminho atual: ['Arad', 'Sibiu', 'Oradea']
  Limite de profundidade 2 alcançado em 'Oradea', retrocedendo.
Expandindo nó: Fagaras (Profundidade: 2). Caminho atual: ['Arad', 'Sibiu', 'Fagaras']
  Limite de profundidade 2 alcançado em 'Fagaras', retrocedendo.
Expandindo nó: RimnicuVilcea (Profundidade: 2). Caminho atual: ['Arad', 'Sibiu', 'RimnicuVilcea']
  Limite de profundidade 2 alcançado em 'RimnicuVilcea', retrocedendo.
Falha! Objetivo 'Bucuresti' não encontrado.

Não foi possível encontrar um caminho (limitado a profundidade 2) de Arad para Bucuresti.
```

Observe que a DFS pode encontrar um caminho diferente da BFS, e sua ordem de exploração é visivelmente diferente, explorando um ramo profundamente antes de retroceder. No exemplo sem limite, o caminho encontrado é mais longo em termos de passos que o caminho encontrado pela BFS.

Propriedades da DFS

- **Completude:** Não é completa. Se o espaço de estados contiver caminhos infinitos ou ciclos e o algoritmo não tiver um mecanismo para evitar visitá-los repetidamente (como o conjunto `visited` que adicionamos), a DFS pode ficar presa em um desses caminhos e nunca encontrar o objetivo, mesmo que ele exista. A versão com limite de profundidade (Depth-Limited Search) pode ser completa se o limite for maior que a profundidade do objetivo.
- **Otimidade:** Não é ótima. A DFS pode encontrar um caminho para o objetivo, mas não há garantia de que será o caminho mais curto ou de menor custo, pois ela se aprofunda em um ramo e pode encontrar um objetivo "longe" antes de explorar caminhos "curtos" em outros ramos.
- **Complexidade de Tempo:** $O(b^m)$, onde `b` é o fator de ramificação e `m` é a profundidade máxima do espaço de estados. No pior caso, a DFS pode ter que explorar toda a árvore de busca até a profundidade máxima.

- **Complexidade de Espaço:** $O(bm)$ para pesquisa em grafo (se armazenar o conjunto de visitados) ou $O(dm)$ para pesquisa em árvore (onde `d` é a profundidade do nó atual). Em geral, é muito mais eficiente em termos de memória do que a BFS, pois armazena apenas o caminho atual e os nós não expandidos em um único ramo.

Comparação entre BFS e DFS

Característica	Busca em Largura (BFS)
Ordem de Exploração	Nível a nível (explora todos os nós na profundidade `k` antes de ir para `k+1`)
Estrutura de Dados	Fila (Queue - FIFO)
Completude	Sim (se `b` é finito)
Otimidade	Sim (se custos uniformes)
Complexidade de Tempo	$O(b^d)$
Complexidade de Espaço	$O(b^d)$ (muito alta para `d` grande)
Vantagens	• Encontra a solução mais rasa/curta primeiro • Completa e ótima para custos uniformes
Desvantagens	Exige muita memória para grandes espaços de estados

Considerações Práticas

- **Escolha do Algoritmo:**
  - Use **BFS** quando você precisa encontrar o **caminho mais curto** (em termos de número de passos) e o fator de ramificação e a profundidade da solução não são excessivamente grandes, ou quando a memória não é uma restrição severa.
  - Use **DFS** quando o **espaço de estados é muito profundo ou infinito**, e você está mais interessado em *encontrar qualquer solução* rapidamente, e a profundidade da solução não é crítica. Também é útil quando a memória é um recurso limitado. Para garantir completude em DFS, é comum usar uma variação como a **Busca em Profundidade Iterativa (Iterative Deepening Depth-First Search - IDDFS)**, que combina as vantagens de ambas (completude e otimalidade da BFS com a eficiência de memória da DFS).
- **Evitando Ciclos:** Para problemas em grafos (onde um estado pode ser alcançado por múltiplos caminhos), é crucial manter um registro dos nós já visitados para evitar ciclos e reprocessamento desnecessário. Ambos os exemplos em Python incorporam essa prática.

Verificação de Aprendizagem

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

1. **Definição de Problema de Busca:** Considere o problema de resolver um quebra-cabeça de 8 peças (um tabuleiro 3x3 com 8 peças numeradas e um espaço vazio, onde as peças podem ser movidas para o espaço vazio). Defina os elementos de um problema de busca para este quebra-cabeça:

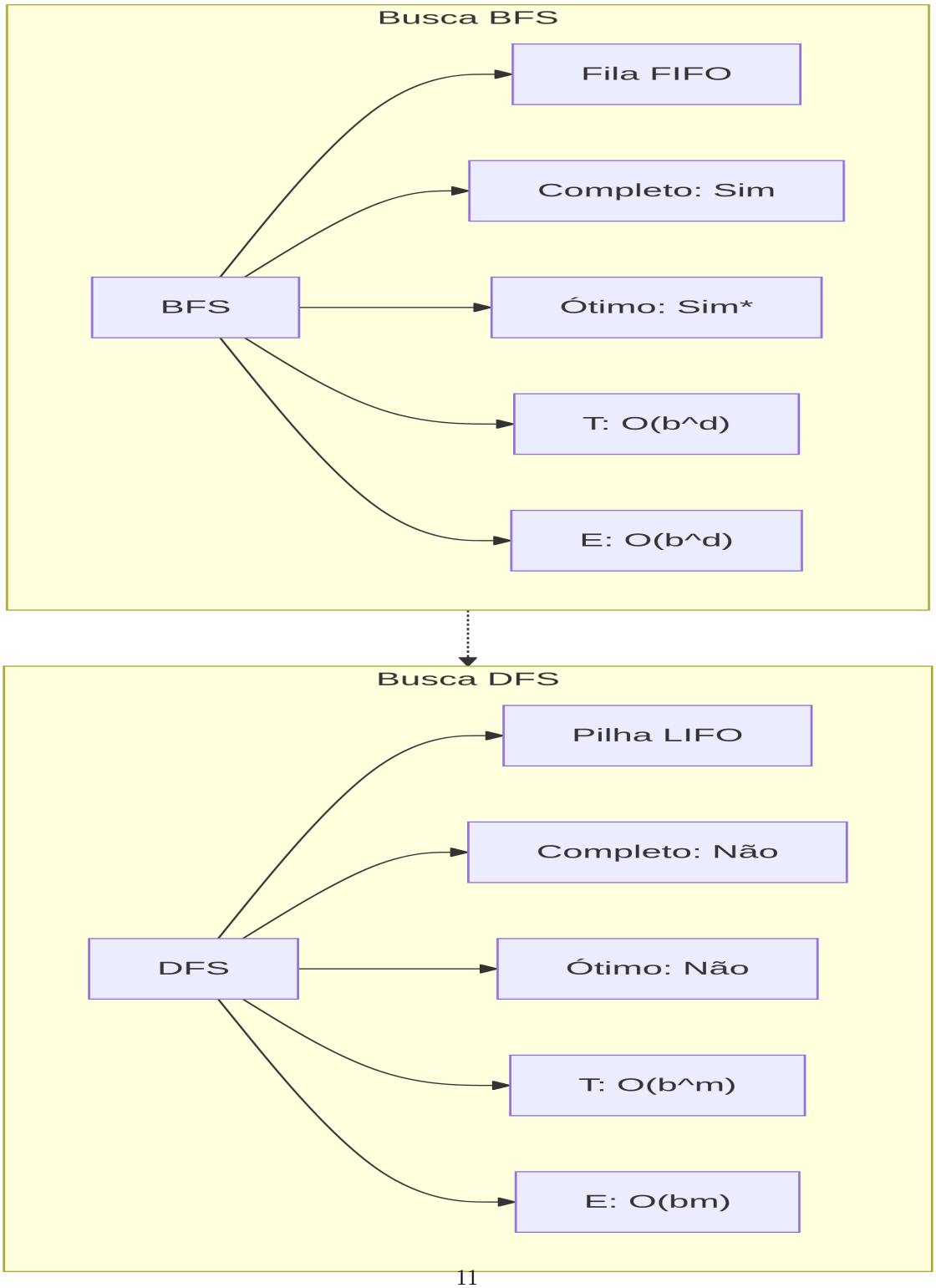

Figure 3: Comparação entre BFS e DFS

a) Estado Inicial
b) Ações (Operadores)
c) Função de Transição de Estado
d) Teste de Meta
e) Custo do Caminho

2. **BFS vs. DFS - Análise:**

   a) Em qual cenário a Busca em Largura (BFS) seria mais vantajosa do que a Busca em Profundidade (DFS) para encontrar um caminho para um objetivo? Justifique.
   b) Em qual cenário a Busca em Profundidade (DFS) seria mais vantajosa do que a Busca em Largura (BFS)? Justifique.
   c) Um problema tem um espaço de estados muito profundo e ramificado. Qual algoritmo você escolheria para encontrar uma solução, considerando a limitação de memória? Explique porquê.

3. **Implementação e Análise:**

   Considere o seguinte grafo simples Figure 4:

a) Represente este grafo em Python como um dicionário de adjacências.
b) Utilizando sua implementação da função `bfs` (ou adaptando-a), execute uma busca de `A` para `F`. Anote a ordem dos nós expandidos e o caminho encontrado.
c) Utilizando sua implementação da função `dfs` (ou adaptando-a), execute uma busca de `A` para `F`. Anote a ordem dos nós expandidos e o caminho encontrado.
d) Compare os caminhos encontrados pela BFS e DFS. Qual deles encontrou o caminho "mais curto" (em número de passos)? Isso confirma as propriedades dos algoritmos?

Referências Bibliográficas

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

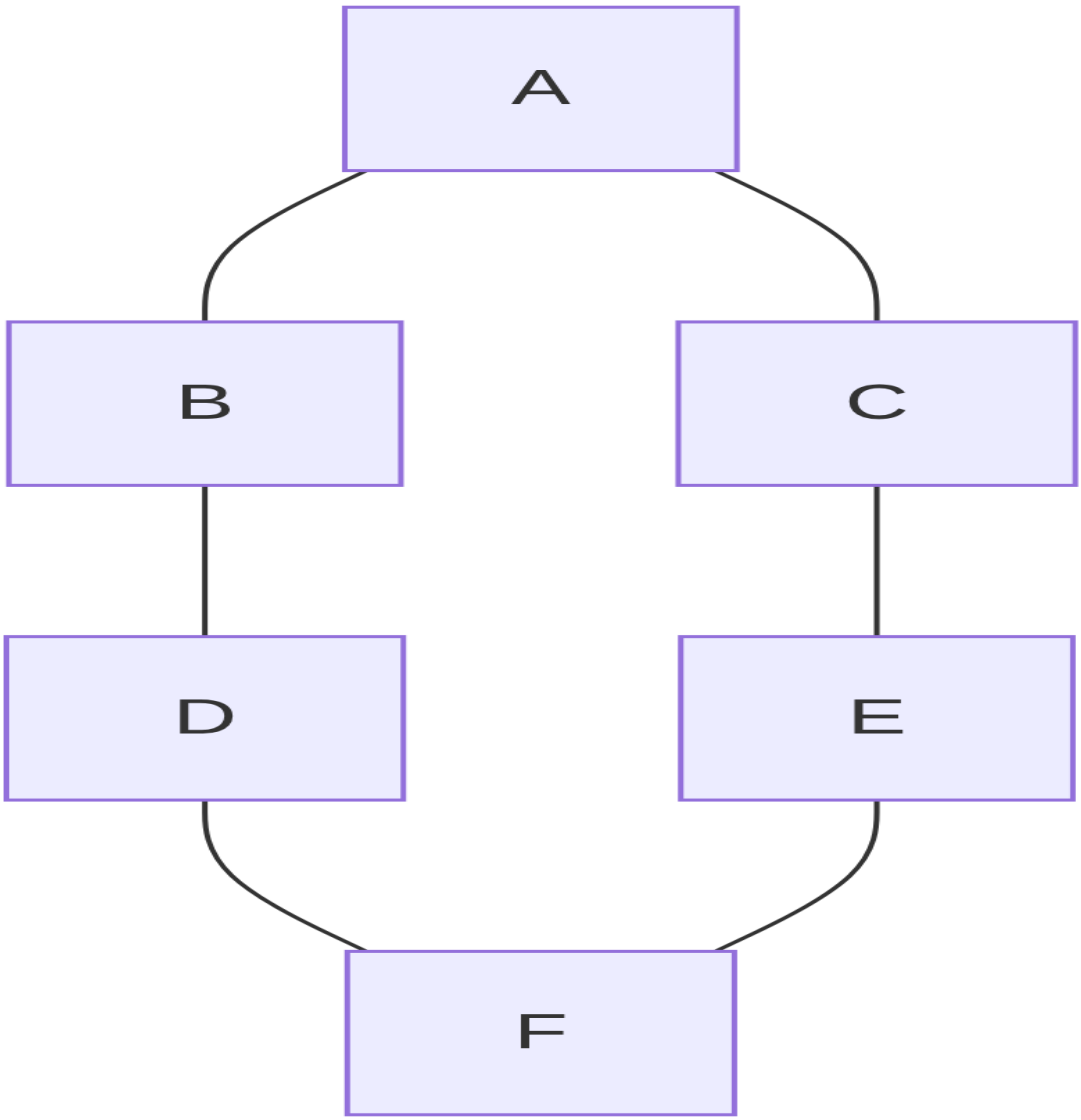

Figure 4: Representação de Problemas para Busca